Andrei Ershov
Fourth International Conference

Perspectives of System Informatics

Preliminary Proceedings

2-6, July, 2001,
Akademgorodok,
Novosibirsk, Russia

PSI'01

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*
12.07.2001

2. REPORT TYPE
conference proceedings

3. DATES COVERED *(From - To)*
January 2001 - July 2001

4. TITLE AND SUBTITLE
Andrei Ershov Fourth International Conference
PERSPECTIVES OF SYSTEM INFORMATICS

5a. CONTRACT NUMBER

5b. GRANT NUMBER
N00014-01-1-0039

5c. PROGRAM ELEMENT NUMBER

5d. PROJECT NUMBER
01PR01132-00

5e. TASK NUMBER

5f. WORK UNIT NUMBER

6. AUTHOR(S)
Dines Bjorner,
Manfred Broy,
Alexandre Zamulin

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
A.P. Ershov Institute of Informatics Systems.
6, Akad. Lavrenitiev pr.,
630090, Novosibirsk, Russia

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Naval Research,
Microsoft Research,
Russian Foundation For Basic Research,
xTech Software Development Company

10. SPONSOR/MONITOR'S ACRONYM(S)
ONR, MR, RFBR, xTech

11. SPONSORING/MONITORING AGENCY REPORT NUMBER

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; distribution is Unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Extended abstracts of the papers
presented at the Andrei Ershov Fourth International Conference
PERSPECTIVES OF SYSTEM INFORMATICS

15. SUBJECT TERMS
system informatics, computer science, software construction

16. SECURITY CLASSIFICATION OF: a. REPORT	b. ABSTRACT	c. THIS PAGE	17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
UNCLASSIFIED			SAR	288	Alexandre Zamulin
					19b. TELEPONE NUMBER *(Include area code)* +7(3832) 396258

PERSPECTIVES OF SYSTEM INFORMATICS

Andrei Ershov Fourth International Conference
2–6 July 2001, Novosibirsk, Akademgorodok, Russia

Sponsored by:

Russian Foundation for Basic Research
Office of Naval Research
Microsoft Research

Organized by:

A. P. Ershov Institute of Informatics Systems,
Siberian Branch of Russian Academy of Sciences

Graphic design:

xTech Ltd.

UDK 519.6

Perspectives of System Informatics (Proceedings of Andrei Ershov Fourth International Conference). — Novosibirsk: A. P. Ershov Institute of Informatics Systems, 2001. — 296 p.

The volume comprises the papers presented at Andrei Ershov Fourth International Conference "Perspectives of System Informatics" held in Akademgorodok (Novosibirsk, Russia), July 2–6, 2001.

Various problems of theoretical computer science, programming methodology and artificial intelligence are considered in the papers.

The book is addressed to specialists in theoretical and systems programming and new information technologies.

This work relates to Department of Navy grant N 00014-01-1-0039 issued by the Office of Naval Research. The United States Goverment has a royalty-free license throught the world in all copyrightable material contained herein.

FOREWORD

The volume comprises extended abstracts of the papers selected for the presentation at the Fourth International Andrei Ershov Memorial Conference "Perspectives of System Informatics", Akademgorodok (Novosibirsk, Russia), July 2–6, 2001. The main goal of the conference is to give an overview of research directions which are decisive for the growth of major areas of research activities in system informatics.

The conference is held to honor the 70th anniversary of the late Academician Andrei Ershov (1931–1988) and his outstanding contributions towards advancing informatics. It is the fourth conference in the line. The First International Conference "Perspectives of System Informatics" was held in Novosibirsk, Akademgorodok, May 27–30, 1991, the second one in June 25–28, 1996, the third one in July 6–9, 1999. The three conferences gathered a wide spectrum of specialists and were undoubtedly very successful.

The fourth conference includes many of the subjects of the previous ones, such as theoretical computer science, programming methodology, and new information technologies, which are the most important components of system informatics. The style of the third conference is preserved to a certain extent: a considerable number of invited papers in addition to contributed regular and short papers.

This time 73 papers were submitted to the conference by researchers from 19 countries. Each paper was reviewed by three experts, at least two of them from the same or closely related discipline as the authors. The reviewers generally provided high quality assessment of the papers and often gave extensive comments to the authors for the possible improvement of the presentation. As a result, the Programme Committee has selected 26 high quality papers as regular talks and 22 papers as short talks. A broad range of hot topics in system informatics is covered by five invited talks given by prominent computer scientists from different countries.

To celebrate the 70th anniversary of the late Academician A. P. Ershov, a special memorial session is organized. It includes two invited talks and a number of short informal communications. The invited talks are given by two prominent Russian computer scientists who worked either side by side with A.P. Ershov or in closely related area.

Andrei P. Ershov was a man for all seasons. He commanded universal respect and received affection all over the world. His view of programming was both a human one and a scientific one. He created at Akademogorodok a unique group of scientists — some now in far away regions of the world: a good example of "technology transfer", although perhaps not one that too many people in Russia are happy about.

Many of his disciples and colleagues continue to work in the directions initiated or stimulated by him, at the A. P. Ershov Institute of Informatics Systems named after him, which is the main organizer of the conference.

We are glad to express our gratitude to all the persons and organizations who contributed to the conference — to the sponsors for their moral, financial and organizational support, and to the members of local Organizing Committee for their mutual efforts towards a success of this event. We are especially grateful to N. Cheremnykh for her selfless labor when preparing the conference.

July, 2001

D. Bjorner,
M. Broy,
A. Zamulin

CONFERENCE CHAIR

Alexander Marchuk (Novosibirsk, Russia)

PROGRAMME COMMITTEE CO-CHAIRS:

Dines Bjørner (Lyngby, Denmark)
Manfred Broy (Munich, Germany)
Alexandre Zamulin (Novosibirsk, Russia)

LOCAL ORGANIZING COMMITTEE

Sergei Kuznetsov
Gennady Alexeev
Alexander Bystrov
Tatyana Churina
Vladimir Detushev
Olga Drobyshevich
Vera Ivanova
Sergei Mylnikov
Elena Okunishnikova
Vladimir Sergeev
Anna Shelukhina
Irina Zanina

INVITED SPEAKERS

Rimma Podlovchenko (Moscow, Russia)
Igor Pottosin (Novosibirsk, Russia)
Egidio Astesiano (Genova, Italy)
Jan Friso Groote (Eindhoven, Netherlands)
Yuri Gurevich (Redmond, USA)
Bertrand Meyer (Santa Barbara, USA)
Peter Mosses (Aarhus, Denmark)

CONFERENCE SECRETARY

Natalia Cheremnykh (Novosibirsk, Russia)

PROGRAMME COMMITTEE

Janis Barzdins (Riga, Latvia)
Frédéric Benhamou (Nantes, France)
Mikhail Bulyonkov (Novosibirsk, Russia)
Gabriel Ciobanu (Iasi, Romania)
Piotr Dembinski (Warsaw, Poland)
Alexander Dikovsky (Nantes, France)
Uwe Glässer (Paderborn, Germany)
Victor Ivannikov (Moscow, Russia)
Philippe Jorrand (Grenoble, France)
Leonid Kalinichenko (Moscow, Russia)
Alexander Kleschev (Vladivostok, Russia)
Gregory Kucherov (Nancy, France)
Sergei Kuznetsov (Moscow, Russia)
Alexander Letichevski (Kiev, Ukraine)
Giorgio Levi (Pisa, Italy)
Dominique Mery (Nancy, France)
Bernhard Möller (Augsburg, Germany)
Hanspeter Mössenböck (Linz, Austria)
Ron Morrison (St. Andrews, Scotland)
Valery Nepomniaschy (Novosibirsk, Russia)
Peter Pepper (Berlin, Germany)
Francesco Parisi-Presicce (Rome, Italy)
Jaan Penjam (Tallinn, Estonia)
Alexander Petrenko (Moscow, Russia)
Jaroslav Pokorny (Prague, Czech Republick)
Wolfgang Reisig (Berlin, Germany)
Viktor Sabelfeld (Karlsruhe, Germany)
Don Sannella (Edinburgh, Scotland)
Vladimir Sazonov (Manchester, UK)
David Schmidt (Manhattan, USA)
Igor Shvetsov (Novosibirsk, Russia)
Sibylle Schupp (Troy, USA)
Nicolas Spyratos (Paris, France)
Lothar Thiele (Zurich, Switzerland)
Alexander Tomilin (Moscow, Russia)
Enn Tyugu (Stockholm, Sweden)
Andrei Voronkov (Manchester, UK)
Tatyana Yakhno (Izmir, Turkey)

Additional Referees

S. Ambroszkiewicz
M. A. Bednarczyk
A. Bossi
O. Bournez
M. Breitling
D. Cansell
S. Chernonozhkin
D. Colnet
M. Dekhtyar
H. Ehler
Th. Ehm
G. Ferrari
J. Fleuriot
J. Freiheit
K. Freivalds
W. Grieskamp
G. Hamilton
P. Hofstedt
P. Hubwieser
P. Jackson
P. Janowski
N. Jones
G. Klein
K. Korovin
A. Kossatchev
P. Lescanne
F. Levi
I. Lomazova
P. Machado
A. Masini
E. Monfroy
D. von Oheimb
P. Poizat
M. Proietti
A. Rabinovich
A. Riazanov
J. Ruf
Ch. Salzmann
F. Saubion
W. Schulte
B. Schaetz
N. Shilov
F. Spoto
N. Tillmann
J. Tucker
M. Turuani
M. Valiev
W. Vogler
J. Winkowski
J. Vain
M. Veanes

TABLE OF CONTENTS

Memorial session

Computing and Algorithms

Logical Methods

Verification

Program Transformation and Synthesis

Semantics & Types

Processes and Concurrency

UML Specification

Petri Nets

Testing

Software Construction

Data & Knowledge Bases

Logic Programming

Constraint Programming

Program Analysis

Language Implementation

A. P. Ershov — a Pioneer and Leader of Programming in Russia

Igor V. Pottosin

A. P. Ershov Institute of Informatics Systems
Siberian Division of the Russian Academy of Sciences
6, Acad. Lavrentiev ave., 630090, Novosibirsk, Russia
e-mail: ivp@iis.nsk.su

Andrei Ershov belonged to the first generation of native programmers. He was among the first University graduates in programming (Moscow State University, 1954). Programming as a profession appeared two years earlier and formed from professional mathematicians and physicists. Taking into account the fact that Ershov became a programmer even in his student years, it is possible to say that he shared the way of programming as a profession and scientific discipline.

Being a pioneer of programming, he passed through all stages of evolution of programming — from a tool for solution of numerical problems to formation of the first independent research fields in programming, such as compilers and languages, operating systems and theoretical models of programs. Like all programmers of the first generation, Ershov has felt all the difficulties and problems connected with formation of a new scientific direction — it was necessary to prove that this direction has the right to exist, has its own scientific value and its problems are as important and essential as the foundations of already formed scientific disciplines. This can be illustrated by the hard history of Ershov's works on operator algorithms, one of the models of program schemata and difficulties with the Alpha-project.

A. Ershov was not just one of the participants of the formation process of a new discipline, he became one of its leaders. His leading role in this direction is out of discussion. It is sufficiently to note the importance of his works and results for self-identification of the new scientific direction.

Ershov was one of the creators of the compilation theory and methodology — the initial research area in programming. Creation of a general, language-independent compilation scheme, the concept of internal representation abstracted from semantic properties of a program, creation of a number of techniques, such as hash functions, memory allocation technique and so on — such were his results in this field. He is the author of the first monograph on program compilation (A.P. Ershov. Programming Program for the BESM Computer. Pergamon Press, London, 1959).

He made an essential step in post-Algol evolution of programming languages: the Alpha-language, an extension of Algol-60, had such properties as multi-dimension variables, various do-statements, initial values and so on. The Sigma language proposed by Ershov was an example of a language kernel extended by substitution mechanism.

The foundation of programming originated from the experience of implementation of real programming systems, was based on this experience. Ershov's leadership was also evident in the fact that he was either an initiator or a supervisor (or both) of a number systems of fundamental importance each of which was based on new ideas and approaches. The most important of them are as follows: the Alpha-project — the first programming system for Algol-like language with high-level program optimization; Aist-0 — a multiprocess and multiuser system with rich multifunctional software; the Beta-project — a multilanguage compiling system with implementation of popular programming languages (Pascal, Modula-2, Simula-67, Ada); the programming system Setl — implementation of one of the first specification language; the workstation Mramor — hardware and software support for publishing activity; the programming system Shkol'nitza — a methodologically grounded tool for school training in programming.

A. Ershov was one of the founders of the program schemata theory (see [1]). It is important to say that he always saw the relation between programming theory and programming practice. The examples of this are application of his memory allocation theory to implementation of memory optimization in the Alpha-system, initialization of research in constructing program models oriented to justification of program optimizations, the development of parallel program models as a part of general research in multiprocessing of Aist-0, and so on.

Mixed computation concept is one of his main results in this symbiosis of theory and practice. This concept, proposed by Ershov as a fundamental one for creation of language processors, became the basis for a number of real specialization systems for imperative and declarative languages.

One of the main problems of the new direction is training of professionals and researchers. Ershov was a pioneer in this field too. His great efforts in foundation of educational informatics, its methology, writing manuals and programming systems for education were very important. He was an absolute leader of this activity in our country.

He had a great influence on the spirit of this new field, its ethics, its professional specific. The social image of programming in our country was formed by activity of such organizations as the Commission on System Software, the Committee on Programming Systems and Languages, and the Council on Cybernetics headed by Ershov. His well-known papers "Two faces of programming", "Aesthetics and the human factors of programming", and "Programming, the second literacy" have defined the spirit and specific of a new kind of activity very brightly and clearly.

References

1. R.I.Podlovchenko. A.A.Liapunov and A.P.Ershov in the theory of program schemes and the development of its logic concepts.
2. D.Bjorner, V.Kotov (Eds). Images of Programming / North-Holland, 1991.
3. V.Kotov, I.Pottosine. Andrei P.Ershov 1931-1988 // Theoretical Computer Science, Vol. 67, 1, 1989, P. 1–4.
4. W.M.Turski. Obituary: Andrei Petrovich Ershov // IEEE Annals of the History of Computing, Vol. 13, 2, 1993, P. 55–58.

A. A. Lyapunov and A. P. Ershov in the Theory of Program Schemes and the Development of Its Logic Concepts

Rimma I. Podlovchenko

Research Computing Center
Moscow State University, Moscow, RU-119899, Russia
e-mail: rip@vvv.srcc.msu.ru

Abstract. The aim of this paper is to survey the advent and maturation of the theory of program schemes, emphasize the fundamental contributions of A. A. Lyapunov and A. P. Ershov in this branch of computer science, and discuss the main trends in the theory of program schemes.

This paper was written in memory of Andrey Petrovich Ershov, who exerted great influence on the development of theoretical programming, the theory of program schemes specifically. The choice of this section for discussion does not only display the authors taste. The main thing is that the concepts laid in the foundation of the theory of program schemes in the years of its coming into being, which were actively introduced by A. P. Ershov, were consolidated in subsequent years along the trend predicted by him. It was intended that this paper would elucidate the facts.

Scientific preferences of Andrey Petrovich were formed in the 50-es of the past century. The years were commemorated by the appearance and development of domestic electronic computers. There was a need in specialists for their designing and servicing.

Moscow State University responded to the call at once. In 1950 the chair of computing mathematics was set up in the faculty of mechanics and mathematics. Teaching of students in numerical analysis was one of the tasks set for the chair. Another task consisted in preparation of the students for using the newly born computers. In contrast to the first one, the task did not have any clear-cut outlines of solution. Initially, it was assumed that the use of computers for solving mathematical problems would necessitate detailed knowledge of the machine design. It was reflected in the choice of disciplines included in the curriculum for those studying in the chair, namely: radio engineering and electronics, electrical engineering, theory of mechanisms and machines, computing machines and instruments, drawing. The subjects enumerated replaced largely such disciplines as the set theory, higher classes of algebra, functional analysis, mathematical logic (the theory of algorithms was not taught yet in those years).

Naturally, in due course the things that were actually indispensable for the graduates of the chair were determined and the curriculum got rid of unnecessary subjects, whereas the mathematical foundation was restored.

Andrey Ershov became a student of the chair of computing mathematics in 1952. He was lucky, as the gaps in his mathematical education in the years of his studentship were eliminated with assistance of Alexey Andreevich Lyapunov, who assumed supervision over Andrey Ershov post graduate education. Alexey Andreevich drawn up a program of qualifying examination for the candidates degree satiated in mathematics and strictly followed its implementation advising personally on the subjects included in the program.

Let us go a couple of years back to 1952, when Alexey Andreevich took the post of professor in the chair of computing mathematics. The event is noteworthy, as scientific life in the chair livened up a lot. Andrey Ershov was then the fourth year student.

His enthusiasm and convictions helped him to turn many of students in the chair into his belief in extraordinary future that lied ahead for the machines and programming. In 1952/1953 academic year Alexey Andreevich read the famous course of lectures Principles of Programming (the relevant materials in a somewhat revised form were published only in 1958 [1]) to the students in the chair. It was the first in the country course in programming that played the fundamental role in the development of a new branch of knowledge, i.e. programming.

A new view on formalization of the concept of algorithm as such was presented, proceeding from convenience of its use when solving practical problems. Meanwhile, the already existent formalizations of the algorithm concept (such as Turings machines, Markovs normal algorithms) were aimed exceptionally at studying the nature of computations rather than practical application. In the course read by Alexey Andreevich a programming language was suggested, which was precursor of the currently used high-level languages; the language was called the operator language. Its introduction made it possible to describe techniques of programming. The operator language and the relevant programming techniques were integrated under the name of the operator method.

The operator language was not formalized. However, the problems of programming could be actually discussed, i.e. for the first time programming was treated as a branch of science with its own problems. Two problems were named by Alexey Andreevich as the main ones, specifically:

- automation of making up programs;
- optimization of programs that were initially made up.

The problems were considered mutually interrelated, though each of them could be studied individually.

Alexey Andreevich attracted students in the chair, Andrey Ershov among them, for coping with the first task. He offered that the operator language is used as support one. Construction of the so-called programming program was planned; the program receiving in its entry an algorithm in the operator language was to transform it into the program executing the algorithm. Conceptually, the programming program was to be assembled from blocks performing individual functions. Andrey Ershov was entrusted with construction of arithmetic block.

The work was the initial step in the studies relating to construction of translators, the studies that ran all through Andrey Petrovich subsequent activities in programming. The works in this trend are enumerated in [2]. In the introductory article by I.V. Pottosin [2] the evolution of his ideas and techniques for constructing the translators is described. It is worth noting that already in that initial work the idea arose that the memory in the programs should be saved (refer to [3]). It was the first manifestation of the global intention to include techniques for optimization of the made up programs in the process of translation.

But let us recall the old days when approaches to coping with the second task were groped for. Alexey Andreevich assumed that, first of all, formalization of the operator language is necessary, taking a full enough account of actual program properties, and then the place occupied by the formalized programs in the series of other algorithms definitions shall be ascertained. He entrusted his post graduate Andrey Ershov with this work, considering it as one of initial stages of theoretical studies in programming.

Mathematical solution of the program optimization problem shall actually rely on strict definition of the program as such, its structure and functions, specifically. Only then one can speak of the function realized by the program and, accordingly, introduce the concept of program equivalence by using the requirement of coincidence of the functions realized by the programs. Optimization of a program is performed by means of its equivalent transformations (e.t.), i.e. transformations, which retain the function realized by the program. Hence, the problem of development of the formalized programs e.t. is brought to the forefront.

A. A. Lyapunov saw one of possible ways of its solution by constructing e.t. using not the programs as such, but their models, i.e. program schemes. The logic schemes of the programs he considered in two ways: as an algorithm description and as an algorithm scheme description. In the first case all operators used in the logic scheme and logic conditions are made specific. In the second case their specific definition is absent, only the places occupied by them being fixed. The logic scheme interpretation as an algorithm scheme resulted in a theoretical concept that was named the Yanov schemes.

But let us go back to the problem that faced A. P. Ershov. Intuition prompted that formalization of a program is a new definition of an algorithm, by which all computable functions will be realized. It was the conclusion made by A.P. Ershov [5], [6]. The result obtained placed the proposed formalization of a program among other formalizations of an algorithm, the mere fact of it being important. Besides, he changed the attitude towards the problem of e.t. development using their schemes. The point is that when mathematical models are considered, usually an e.t. system that is complete in a given class of objects is constructed. Let it be a class of programs. The *completeness of an e.t. system* implies that for any two equivalent programs belonging to the given class, there exists a finite chain of transformations belonging to the system that transforms one program into the other. Clearly, this problem has the trivial decision it is the system consisting of all pairs of equivalent programs belonging to the given class. But this decision is rejected.

As solvable e.t. systems are of practical interest, their search ranks among *e.t. problems*. But then the necessary condition for its positive solution is decidability of the equivalence problem (it consists in construction of algorithm that recognizes the equivalence of programs). But if all computable functions are realized by the programs, the equivalence problem is not decidable for them, the fact being mentioned in [4]. Then turning to program schemes for the development of e.t. systems takes on the status of practically necessary task.

The task mentioned was considered for the first time by A. A. Lyapunov disciple — Yu. I. Yanov in [7]. The study was the first one in the inceptive theory of program schemes. Let us dwell on the results.

We will describe the structure of the Yanov scheme by a graph suggested by A. P. Ershov and its function corresponding to [9].

A *program scheme* (or simply *scheme*) is described by a finite directed graph. Two nodes of the graph are different from the others: the entry node, which has only one outgoing edge and has no entering edges, and the exit node, which has no outgoing edges. The other nodes of the graph are either transformers or recognizers. Each transformer has one outgoing edge and is associated with an operator symbol from Y. Each recognizer has two outgoing edges with marks 0 and 1 and is associated with a logical variable from P.

An example of a program scheme is depicted in Fig. 1. This scheme corresponds to the algorithm that computes value $n!$ for $n \geq 0$ (see Fig. 2).

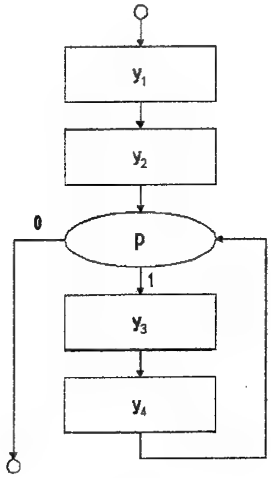


Fig. 1

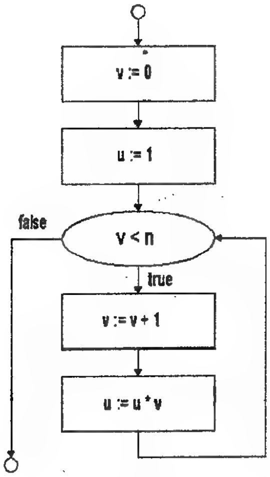


Fig. 2

The functional description of a scheme is related to the process of its execution, which consists in traveling through the scheme accompanied by the accumulation of a chain of operator symbols. The corresponding path is determined by a priori given labeling function, which is defined as follows.

Let

$$X = \{x | x : P \to \{0,1\}\}.$$

The elements of X are sets of values of all logical variables. Words in the alphabet Y will be referred to as *operator chains*. The *labeling function* is a mapping of the set Y^* consisting of all operator chains into the set X. Denote by L the set of all labeling functions.

Let G be a scheme and μ be a function from L. The execution of the scheme G on the function μ begins at the entry node of the scheme with the empty operator chain and consists in tracing the scheme. The passage through a transformer is accompanied by adding from the right an operator symbol corresponding to this transformer to the current chain. The passage through a recognizer does not change the current chain. Let h be the current chain and p be a variable assigned to the recognizer. Then, the value of the variable p is extracted from the set μh, and the tracing is continued along the edge marked by this value. The scheme execution is completed when the process reaches the exit of the scheme. In this case, the scheme G is said *to stop* on μ, and the result of its execution is the operator chain accumulated.

In [7] Yu.I. Yanov considered a parametric set of equivalences of schemes over basis Y, P. Parameter denoted by s was named a *shift distribution* in Y; it induces the set of labeling functions denoted by L_s. By definition schemes $G1, G2$ are equivalent for the given s, if and only if they stop on the same functions from L_s and chains obtained for a given function coincide.

Ground result of [7] is theorem 1.

Theorem 1. *Whatever is a shift distribution in Y for the equivalence induced by them both problems (of equivalence and of e.t.) are decided in the class of schemes over Y, P, that use each operator symbol less than twice.*

Decidability of both problems was proved later for the class of all schemes over Y, P.

Practically at once the question is appropriate: how the above-mentioned equivalences could be interpreted informally. Rutledge [10] was the first one to suggest an answer to the question. Let us illustrate his approach using one equivalence by way of example. It is induced by the shift distribution s, for which $L_s = L$, being called *a strong equivalence.*

We shall introduce two notions: a semantics of basis Y, P and an abstract program induced by them.

A *semantics* of basis Y, P denoted by σ is a complex consisting of:

- a set Ξ_σ; its elements being named as states;
- functions σy: $\Xi_\sigma \to \Xi_\sigma$, $y \in Y$;
- relations σp: $\Xi_\sigma \to \{0, 1\}$, $p \in P$.

The process of executing the scheme G on pair $(\sigma, \xi_0), \xi_0 \in \Xi_\sigma$, begins at entry node of the scheme with the state ξ_0 and consists in tracing the scheme. The passage through a transformer with symbol y is accompanied by transformation of the current state ξ into state $\sigma y(\xi)$. The passage through a recognizer with variable p does not change the current state ξ; the tracing is continued along the edge marked by $\sigma p(\xi)$. The scheme execution is completed when the process reaches the exit of the scheme and then the current state is a result of the execution. Scheme G accompanied by semantics σ is named an *abstract program*; one images the set Ξ_σ into itself. Two abstract programs are *equivalent* if and only if they realize the same function.

Theorem 2. *Two schemes over basis Y, P are strongly equivalent if and only if they induce equivalent abstract program for any semantics of Y, P.*

For the first time the fact was established in [10].

The investigations carried out by Yu.I. Yanov precede the advent of the finite automaton notion. The connection between the Yanov schemes and a finite automata is given in theorem 3.

Theorem 3. *Whatever is a shift distribution in Y, the equivalence induced is reduced to the equivalence of finite automata.*

For a strong equivalence this fact is established in [10]; for the others it follows from the statement: L_s is the regular language.

Investigations aimed at economy of program memory made fundamental contribution into the program scheme theory along with the Yanov schemes. They are considered to the maximum extent by A. P. Ershov [11].

Let us discuss the methodology of the program scheme theory. The thesis formulated by A. A. Lyapunov, i.e. **the program schemes are created for construction of program e.t.**, is the primary one. The logical concepts set forth by A. P. Ershov in [11] and previously in [12] rest on it. The concepts are formulated below.

Concept 1 *Formalization of a program, the description of the program structure and functioning, as well as definition of the program equivalence, is the initial point for constructing the program schemes. When constructing the Yanov schemes (schemes over non-distributable memory, in general) the formalization of program is implied and needs a strong definition. This action is performed in [10] for the Yanov schemes.*

In case of schemes over distributable memory, the programs are preliminarily formalized. For illustration of this position we shall consider the *standard schemes* described in [12]. They are created for the ALGOL-like programs, in which the description of variables is deleted. The programs are constructed over the basis consisting of assignment operators, **go to** operator and Boolean expressions by using all known operations of operator composition except for the procedure operator.

An example of such a program is given in Fig. 2.

Concept 2 *The definition of a program scheme consists in the description of its structure and functioning and the introduction of the scheme equivalence. These components obey the rules:*

a. the structure of the scheme coincides with the structure of the program modeled by the scheme;
b. the equivalence of the schemes implies the equivalence of programs modeled by the schemes.

These conditions follow from the primary thesis. Really, if condition a is executed, then each transformation of the scheme is the transformation of the program modeled by the scheme. Condition b secures the following: if the first transformation is equivalent, then the second one is equivalent, as well.

We shall illustrate the execution of concept 2 for the standard schemes.

Condition a is satisfied. Really, the transfer from a program to a standard scheme is realized by replacement of concrete operations and relations used in basis operators of assignment and, accordingly, in Boolean expressions by *functional* and *predicative* symbols respectively; the first the second symbols retain the number of arguments of operations and relations. The constructions obtained from concrete operators and Boolean expressions are

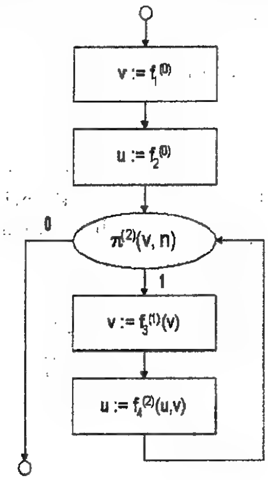


Fig. 3.

named *operators over memory* and *predicates over memory.* The structure of the scheme coincides with the structure of the program inducing the scheme.

Fig. 3 provides the standard scheme constructed for the program depicted in Fig.2.

We shall describe the functioning of the standard scheme. Let i be an interpretation of functional and predicative symbols, replacing concrete operations and relations. The interpretation of i translates the scheme to the *i-program.* The function realized by the i-program is defined by the same rules that define the function realized by the initial program.

Two standard schemes are *equivalent* if and only if for any interpretation i they give i-programs realized by the same function for the given interpretation. And, as there is an interpretation transforming the scheme into the initial program among interpretations i, condition b is satisfied.

The satisfying of concept 2 explains the popularity of standard schemes in the theory of program schemes (see [13]).

Concept 3 *The e.t. problem for schemes is among the leading problems in the theory. This problem is formulated like e.t. problem for programs.*

Concept 4 *The problem of scheme equivalence is the fundamental one.*

Let us remind that decidability of the equivalence problem in a class of schemes is the necessary condition for searching an e.t. system complete in the class.

In [12] the problems facing the theory of program schemes were formulated. We shall dwell on two of them.

I. Research of the equivalence problem for schemes.

The initial point is existence of classes of schemes, for which the equivalence problem is not decidable. This fact was established in [14], [15]. The resulting problem is the search for the classes of schemes, where the equivalence problem is soluble. The approaches to the searching are described in [12]. They consist in a demand to semigroup of basic operators or in a restriction on the scheme structure. Both approaches are obvious. But A.P. Ershov described implicit approach. By this the following is meant. The scheme equivalence considered above is based on the concurrence of the functions realized by the schemes. This equivalence is named the *functional* equivalence. Then the final result of the scheme execution is taken. A.P Ershov introduced in consideration the history of the scheme execution and equivalence relations on the histories. Obviously, the equivalence has a practical value, if it is stronger than functional equivalence (for example, the logic-term equivalence, discussed in [12], satisfies the requirement). In [16] it was ascertained that equivalence stronger than functional is reduced to the functional equivalence in a subclass of the scheme.

We shall discuss problem 1 later.

II. Creation of a suitable apparatus of notions for constructing complete e.t. systems for schemes.

In line with tradition going back to mathematical logic the means used for constructing a complete system are somewhat restricted. A formal calculus is created: his formulae are pairs of scheme fragments. A fragment of the scheme is defined so that the scheme is an individual case of the fragment. Equivalence of schemes is expanded on the set of all fragments. The axioms and the rules of the calculus conclusion are defined so that for any pair $(G1, G2)$ of equivalent schemes belonging to the class considered there exists a derivation of pair $(G2, G2)$ from it.

A.P. Ershov using a graph image of a scheme introduced the notion of a scheme fragment and created a new calculus for decision of e.t. problem in the class of the Yanov schemes. Actually, he brought up the question: the axioms and the rules of conclusion must take into consideration the specificity of scheme transformations. This problem was considered in all the works dealing with e.t. problem. The authors position will be elucidated later.

We shall turn now to the branch of the program scheme theory, where the concepts mentioned above will be supported and developed. By this the algebraic theory of computer program models is meant (theory of program models, for short). The advent of this branch is related to [17]—[19]. At the beginning the theory the program formalization was used, which was previously described as the abstract program. Then the abstract program was expanded by the introduction of subprograms [20]. We shall use the priority formalization.

At first we shall discuss how concept 2 was developed.

The structure of a program scheme coincides with the structure of the Yanov scheme. Hence, condition *a* of concept 2 is met.

In the set of program schemes there is a parametric set of equivalences introduced. It expanded essentially the equivalence set considered by Yu.I. Yanov. Each equivalence is induced now by two parameters, they are:

- equivalence v in the Y^*;
- subset L, where $L \subseteq L$.

(v, L)-*equivalence* of schemes $G1, G2$ is defined as follows: for any function from L each time when one of $G1, G2$ stops on this function, the other one also stops, and the results of their execution are two v-equivalent operator chains.

The set of schemes over Y, P with the given (v, L)-equivalence is named (v, L)-*model of programs*.

Following concept 2, (v, L)-equivalence by definition is useful if it is the *approximating* one, that is to say there is a non-empty set S of semantics of basis Y, P, so that for any schemes $G1, G2$ over basis Y, P the following assumption: "$G1, G2$ are equivalent, if and only if on any semantics from S they are equivalent abstract programs" is true.

In [18] theorem 4 is proved.

Theorem 4. *Semigroup equivalence is a sufficient condition for (v, L)-equivalence of schemes over Y, P to be the approximating one.*

By definition (v, L)-equivalence is the semigroup equivalence, if:

a) v has the property $(*)$: for any operator chains h_1, h_2, h_3, h_4 from Y^*

$$(h_1 \overset{v}{\sim} h_2)\&(h_3 \overset{v}{\sim} h_4) \Rightarrow (h_1 h_3 \overset{v}{\sim} h_2 h_4);$$

b) L consists of v-coordinated functions; such function satisfies the demand: for any h_1, h_2 from Y^*

$$(h_1 \overset{v}{\sim} h_2) \Rightarrow \mu h_1 = \mu h_2;$$

c) L is closed in respect to shift operation, i.e.: whatever are function μ from L and chain h from Y^*, the labeling function μ', where

$$\mu' g = \mu h g,\ g \in Y^*,$$

belongs to L. We interpret $h_1 \overset{v}{\sim} h_2$ as the proposition "h_1, h_2 are v-equivalent".

Note that the equivalences of discrete processors discussed by A. A. Letichevsky in [21] are the semigroup equivalences.

Now let us consider one nontrivial question: how can we relate (v, L)-models and the standard schemes.

Let us consider the programs given in the formalization used in standard schemes. Denote by K the class of such programs constructed over a finite basis of assignment operators and Boolean expressions. By transition from programs of class K to standard schemes corresponding to the programs each assignment operator is replaced by operator over memory, each Boolean expression is replaced by predicate over memory. Denote by K_1 the standard scheme class obtained. If the assignment operators are replaced by operator symbols and the Boolean expressions are replaced by logical variables, then we obtain the Yanov schemes from the programs of K. Denote by K_2 their class. Let us now introduce the correspondence between the operator over memory (the predicates over memory) used in K_1 and the operator symbols (the logical variables) used in K_2 so that their prototypes coincide. In this way the correspondence between the schemes of K_1 and schemes of K_2; is established.

Theorem 5. *It is possible to define (v, L)-equivalence in K_2 so that the proposition: "the schemes from K are equivalent, if and only if the schemes from K_2, corresponding to them, are (v, L)-equivalent" will be true. If the equivalence in K_1 is strict, then the (v, L)-equivalence constructed is the semigroup equivalence.*

Here the equivalence in K_1 is named strict, if it is induced by the program equivalence when the results of the program execution coincide on each variable used in basis of operators and expressions.

Corollary of theorem 5. *The set of semigroup equivalences contains all strict equivalences of standard schemes.*

One of advantages of the semigroup equivalence studied is that they factor out the equivalences of standard schemes. But there are another advantages.

By definition, (v_1, L_1)-equivalence is *approximated* by (v_2, L_2)-equivalence, if the second implies the first. Suppose it takes place. Let us consider a class of abstract programs. Suppose their equivalence is approximated by (v_i, L_i)-equivalences, $i = 1, 2$, and for both equivalences complete e.t. systems exist, T_1 is the system for the first, T_2 is the system for the second equivalence. Both systems are not complete in the class of programs but T_1 is richer than T_2.

Thus, in the set of program models we may improve the e.t. system for programs not leaving this set. The possibility gives rise to the task: to search for sufficient indications for approximating one equivalence by another. The problem mentioned is considered in [22].

Now we turn our attention to the equivalence problem for schemes. The latest survey on this topic was presented by V.A. Zakharov in the conference MCU2001 (International Conference Machines et Calculs Universels. Machines, Computations and Universality, Chisinau, 2001, Moldova) and published in [25]. Hence, we restrict our consideration by the novel approaches to this problem and discuss the most significant results obtained so far.

One of the new aspects in studying the equivalence problem is search for algorithms that besides checking the equivalence of a program scheme do it in a reasonable time. The point is that the early investigations were focused mostly on the decidability/undecidability of the equivalence problem and computational complexity of decision procedures was ignored very often. Decision procedures, whose timed complexity is exponential, of the size of schemes under consideration were regarded as workable though quite inapplicable in practice. Since only those algorithms, whose time complexity is polynomial of the size of inputs are acknowledged to be efficient, the question arises as to whether it is possible to find out such algorithms by revising the known decidable cases and attacking new variants of the equivalence problem.

Nowadays two novel techniques for designing efficient equivalence-checking algorithms are developed. Both methods go back to [24].

The essentials of the first method are presented in Theorem 6 below.

Let us introduce some basic concepts.

It is worth noting that the set Y^* of operator chains along with concatenation operation may be thought of as a finitely generated semigroup. Its elements are generated by $y, y \in Y$; the empty chain λ stands for its neutral element.

Let v be an equivalence relation on Y^*, and L be the set of all v-coordinated functions from L. In this case we will say that (v, L)-equivalence is the *equivalence with respect to* (w.r.t.) *semigroup Y^* supplied with v.*

Semigroup Y^* is called *length-preserving*, if v meets requirement $(*)$ (see theorem 4) and, moreover, whenever h and g are v-equivalent chains, then they have the same length.

Given a length-preserving semigroup of operators Y^* and a chain h from Y^*, we denote by $[h]$ the equivalence class of h, and by $|h|$ the length of h. Clearly, the set

$$E = \{< [h_i], [h_2] > | \, |h_1| = |h_2|, h_1, h_2 \in Y^*\}$$

is also a finitely generated semigroup of operators.

We consider some finitely generated semigroup of operators W, which has $\circ$ for binary operation and e for the neutral element. Suppose that U is a semi-subgroup of W, and w^+, w^* are some distinguished elements in W. Then a quadruple

$$K =< W, U, w^+, w^* >$$

is called a *criterial system for a length-preserving semigroup* Y^*, if there exists an integer k_o and a homomorphism φ from E to U, which satisfy the following requirements:

C1.

$$[h_1] = [h_2] <=> w^+ \circ \varphi(< [h_1], [h_2] >) \circ w^* = e$$

holds for all pairs of $h_1, h2$ from Y^*;

C2. For every w from the $U \circ w^*$ there exist at most k_0 left inverse elements from the $w^+ \circ U$, i.e. the equation

$$w' \circ w^* = e$$

has at most k_0 solutions w' of the form $w^+ \circ u$, where $u \in U$.

Then we arrive at

Theorem 6. *Suppose that a length-preserving semigroup Y^* has a criterial system K such that the identity problem "$w_1 = w_2$?" is decidable in time $t(m)$.Then the equivalence problem for schemes w.r.t. this length-preserving semigroup is decidable in time*

$$c_1 n^2 (t(c_2 n^2) + \log n),$$

where n is the size of schemes to be analyzed, whereas c_1, c_2 are constants that depend on k_0, the number of elements in Y and P, and on homomorphism φ.

This theorem, as well as its application to some specific length-preserving semigroup Y^* is presented in [25]. One of such semigroup, namely *free commutative* one, was studied earlier in [24]. A free commutative semigroup Y^* is characterized by the following property: chains h_1 and h_2 are equivalent if for every y in Y the numbers of occurrences of y in h_1 and h_2 is the same. The scheme equivalence w.r.t. free commutative semigroup is decidable in time $cn^2 \log n$, where n is defined as in Theorem 6, whereas c depends on the number of logical variables in P only.

The technique used in [25] is based on the study of algebraic properties of semigroup Y^* supplied with the equivalence.

An alternative approach was introduced by the author [26]. It is based on the computation of invariants for equivalent schemes. By applying this method the following result was obtained in [26].

Theorem 7. *The scheme equivalence w.r.t. semigroup Y^*, which is both left- and right-contracted is decidable in polynomial time.*

By a left-(right-) contracted semigroup we mean any semigroup Y^* supplied with a decidable equivalence v, which in addition to common requirement (*) satisfies the following properties: for any h, h_1, h_2 from Y^*

$$[hh_1] = [hh_2] \Rightarrow [h_1] = [h_2]$$
$$[h_1 h] = [h_2 h] \Rightarrow [h_1] = [h_2].$$

It should be noted that a free commutative semigroup is both left- and right-contracted.

Now we think that the task of attracting attention to new trends in studies of equivalence problem is completed and we turn to the presentation of the latest achievements in the research on equivalent transformations in the framework of program models [26].

As usually, when speaking about complete system of equivalent transformations we mean an formal calculus of scheme fragments, which is complete w.r.t. to some distinguished equivalence on program schemes. By a fragment of program scheme we mean any part of a scheme, whose connection with the rest of the scheme via incoming and outgoing edges is specified. The incoming edges take off from nodes outside the fragment and outgoing edges lead to nodes that are outside the fragment, as well.

Operation of substitution is defined on the set of fragments; let F be a fragment and let F_1 be a subfragment of F; then if a fragment F_2 is coordinated with F_1 (this relation is commutative) the replacement of F_1 with F_2 is admissible; its result is a fragment. Thus, each pair F_1, F_2 of coordinated fragments induces the set of scheme transformations. If one consists of e.t., then F_1, F_2 are named equally useful fragments.

Calculus to be found has one rule of conclusion; it is substitution. Each axiom is formed by a solvable set of pairs of coordinated fragments. A system of equivalent transformations is given by a finite set of axioms and therefore is called finite.

For example, the set of all pairs (F_1, F_2) such that F_i is a fragment, which has neither entry node nor incoming edges, and F_2 is the empty fragment, is an axiom. Replacement of one of such fragments by the other is an equivalent transformation for every equivalence relation alone. Actually, any occurrence of $F_i, i = 1, 2$, has a property: all its nodes are unattainable from the entry and, hence, do not affect the function computed by a scheme.

By using invariants of equivalent schemes we prove the following.

Theorem 8. *If the equivalence of schemes over Y, P is an equivalence w.r.t. semigroup Y^*, which is both left- and right-contracted, then there exists a finite system of equivalent transformations, which is complete in this set of schemes.*

This generalizes many known results on equivalent transformations.

We conclude our survey on the program scheme theory with the following summary: the development of the program scheme theory lends credence to the fruitfulness of its basic concepts. Furthermore, program schemes fit naturally into the row of computational models both by main research problems and by inter-reducibility of these problems.

References

1. A. A. Lyapunov. On logical program schemata. In Problemy kibernetiki, 1958, issue 1, p. 46-74 (in Russian).
2. A.P. Ershov. Selected works. Novosibirsk, Nauka, 1994 (in Russian).
3. A.P. Ershov. On programming of arithmetic operations. Communs. ASM, 1958, v.1, No. 8, p. 3-6.
4. H.G. Rice. Classes of recursively enumerable sets and their decision problems. Trans. Amer. Math. Soc., bf 89, 1953, p. 25-59.
5. A.P. Ershov. Operator algorithms. 1. Basic conceptions. In Problemy kibernetiki. 1960, issue 3, p. 5-48 (in Russian).
6. A.P. Ershov. Operator algorithms. 2. Description of basic constructions of programming. In Problemy kibernetiki, 1962, issue 8. p.211-233 (in Russian).
7. Ya.I. Yanov. On logical algorithm schemata. In Problemy kibernetiki, 1958, issue 1. p. 75-127 (in Russian).
8. A.P. Ershov. Operator algorithms. 3. On operator schemata of Yanov. In Problemy kibernetiki. 1968, issue 20. p.181-200 (in Russian).
9. R.I. Podlovchenko. From schemes of Yanov to the theory of program models. In Mat. voprosy kibernetiki. 1998, issue 7. p. 281-302 (in Russian).
10. J.D.Rutledge. On Ianovs program schemata. J. ACM. 11. 1964. p. 1-9.
11. A.P. Ershov. Introduction to theoretical programming. Moscow. Nauka. 1977 (in Russian).
12. A.P. Ershov. Contemporary state of theory of program schemes. In Problemy kibernetiki. 1973, issue 27, p. 87-110 (in Russian).
13. V.E. Kotov, B.K. Sabelfeld. Theory of program schemes. Moscow. Nauka. 1991 (in Russian).
14. M.S. Paterson. Programs schemata. Machine Intelligence. Edinburgh: Univ. Press. 3. 1968.p.19-31.
15. A.A. Letichevsky. Functional equivalence of finite transformers. II. Cybernetics, 1970, No. 2, p. 14-28 (in Russian).
16. R.I. Podlovchenko. On correctness and essentiality of some Yanov schemas equivalence relations. In LNCS. 32. 1975. p.
17. R.I. Podlovchenko. Hierarchy of program models. Progammirovanie. 1981, No. 2, p. 3- 14 (in Russian).
18. R.I. Podlovchenko. Semigroup models of programs. Programmirovanie. 1981, No. 4, p. 3-13 (in Russian).
19. R.I. Podlovchenko. The program models over structured basis. Programmirovanie. 1982, No. 1. p. 9-19 (in Russian).
20. R.I. Podlovchenko. Recursive programs and hierarchy of their models. Programmirovanie. 1991. No. 6. p. 44-51 (in Russian).
21. A.A. Letichevsky. On the equivalence of automata over semigroup. Theoretic Cybernetics. 6. 1970. p. 3-71 (in Russian).
22. R.I. Podlovchenko, S.V. Popov. The approximating relation on a program model set. Vestnik MGU. 2001, No. 2 (in Russian).
23. V.A. Zakharov. The equivalence problem for computational models: decidable and undecidable cases. In LNCS.
24. R.I. Podlovchenko, V.A. Zakharov. A Polynomial-time algorithm that recognizes the commutative equivalence of program schemes. Doklady Mathematics. Vol.58. No. 2. 1998. p.306-309.
25. V.A. Zakharov. Rapid algorithms for recognizing of equivalence in the class operator program on balanced frames. Mat. voprosy kibernetiki. 1998. issue 7. p. 303-324 (in Russian).
26. R.I. Podlovchenko. On one mass decision of the problem of equivalent transformations for the program schemes. Programmirovanie. 2000. No. 2. p. 66-71; 2001, No. 2. p. 3-11 (in Russian).

Computing and Algorithms

The Abstract State Machine Paradigm: What is In and What is Out

Short Abstract

Yuri Gurevich

Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA
e-mail: gurevich@microsoft.com

The computing science is about computations. But what is a computation? We try to answer this question *without* fixing a computation model first. This brings up additional foundational questions like what is a level of abstraction? The analysis leads us to the notion of abstract state machine (ASM) and to the ASM thesis:

> Let A be any computer system at a fixed level of abstraction. There is an abstract state machine B that step-for-step simulates A.

In the case of sequential computations, the thesis has been proved from first principles; see ACM Transactions on Computational Logic, vol. 1, no. 1 (July 2000), pages 77–111. Of course ASMs are not necessarily sequential. In a distributed ASM, computing agents are represented in the global state. New agents can be created, and old agents can be deactivated. There could be various relations among agents and various operations on agents. The global state is a mathematical abstraction different from the conventional shared memory; it may be, for example, that the agents communicate only by messages. The moves of different agents form a partially ordered set. Concurrent moves cause consistent changes of the global state.

Often a formal method comes with a reasoning system. If this is your idea of a formal method then the ASM approach is not a formal method. It is system informatics where modeling is carefully separated from formal reasoning. Notice that formal reasoning is possible only when the raw computational reality is given a mathematical form; ASMs do the modeling job.

The separation of modeling and reasoning concerns does not undermine the role of reasoning. The ASM approach is not married to any particular formal reasoning system and is open to all of them. It is usual for ASMs to have integrity constraints on states. ASM programs can be enhanced with various pre and post conditions. ASM-based testing can be enhanced with model checking. The most important direct application of ASMs is their use as executable specifications. This makes (totally as well as partially) automated reasoning relevant.

For more information on abstract state machines see the academic ASM website
`http://www.eecs.umich.edu/gasm/`.

On Algorithmic Unsolvability

Svyatoslav S. Lavrov

Institute of Applied Astronomy
Russian Academy of Sciences
8, Zhdanovskaya Str., St.Perersburg, 197110, Russia
e-mail: ssl@lvr.usr.pu.ru

Abstract. Many computational problems are algorithmically unsolvable. How well this fundamental assertion of computability theory is based — that is the main question considered in the paper from a programmer's view.

Introduction — Where Does the Border between Natural and Formal Languages Lie?

To begin with — a comment to the Russel's 'village barber' paradox. A man has many aspects. When at home he eats, sleeps, in the morning washes and possibly shaves himself and after breakfast goes to the work. At work he being the barber receives his clients, cuts their hair and shaves them. He would violate his promise only if he sat in the barber's chair and simultaneously stood nearby and shaved the man sitting there.

One may oppose that all said above is a game with notions taken from the human mode of life and reflected in natural language. However G. Cantor himself expressed the concepts of the set and membership using the words like "collection", "intuition", "intellect", "the whole (indivisible)", which differ from the common ones maybe by a slightly higher style. He simply has had no other means just as we have not got them still. The chance for a set to be its own member is by no means better than for a barber to receive himself as a client.

The attempts taken at the first half of the XX century to remove contradictions from the set theory have led to the seemingly successful creation of the axiomatic set theory. The proper classes introduced in the theory serve as substitute for all ugly sets (undesirable barbers).

The main trouble with the axiomatic approach is in lacking of a model for the whole set theory. The theory with proper classes may be considered as a metatheory for the one of the common sets. It contains the class playing the role of a subject domain for the latter theory. What may serve however as such a domain for the metatheory? In this vicinity the border lies between natural language together with common sense and formal means for expressing the scientific concepts.

1 The Human Factor

A human whatever his occupation may be hardly has no personal view on the subject of the occupation. E. g. every researcher has probably his own concept of the continuum: either it is a set "composed" in a manner from all real numbers or rather a kind of a memory where all rational numbers are written and there are places in between for the other, irrational, numbers when they come into consideration.

If the researcher tries to formulate this concept then a description of a countable set arises — no other sets may be described. He may tell to his colleague the description and the latter says: "But this description is not full, since departing from it I may point out an object of the same kind differing from all objects falling under the description" (here lies the essence of the diagonal method which serves as a mean to prove many fundamental mathematical assertions). The reaction may vary from "Yes, you are possibly right" to "Nobody is interested in your object, in any case not me". This may be answered differently too, but there still remains the problem of constructibility and convincingness of the diagonal method or broader — of a personal view on the science and its substances.

2 Abstract Computation

In the traditional computability theory (cf., e. g., [2, ch. 5]) a process of *computation* starts from some *input data* and ends in a favourable case with supplying of an appropriate *result*. An *abstract machine* is described which works over *data*. The process is usually divided in *steps*. On each step the machine executes just one *elementary action*, guided by a *rule* selected from a fixed finite collection. The *current* data are used — those available at the step beginning. The input data of the process serve as the current ones for the very first step.

A check is made too on each step whether the process is completed with no guarantee that it occurs at some time. Even a man observing the machine functioning (what is not forbidden but with no right to interfere) hardly if at all can in general case predict the future development of events.

The decsription of the sequence of rules which leads the machine to solving a *problem*, i. e. to getting a result tied in a specified manner with the input data, is called an *algorithm* of the problem solving. The algorithm may be considered as a composition of a number of *functions.*

Any abstract machine implements the idea of the *potential infinity.* Thus from any natural number n it is possible to pass to the number $n+1$. Regardless how many words in a finite alphabet are constructed there is a possibility to build a new word at any time.

The *recursion* applies the same idea to function computation. If the required result is not yet got then the function may — directly or via some other functions — call itself to continue the computation.

3 Is It Possible to Establish the Bound of the Recursion Depth?

This question occupies one of the leading places in the computability theory. Let us try to find a sufficiently general answer. To this goal we need two auxiliary functions. The apparently recursive function

$$B() = \text{if } T \text{ then } T \text{ else } B()$$

breaks immediately from the loop of recursive calls with the value T ('true'), while the other one

$$C() = \text{if } T \text{ then } C() \text{ else } T$$

sticks in the loop forever.

Let us assume that a function $S(A, X)$ with two parameters: a function A and its argument X — may be described and that it supplies the value T, if the evaluation of $A(X)$ ends successfully and the value F ('false') — otherwise. The traditional computability theory asserts that such an assumption leads to a contradiction.

The assertion has mainly the following proof. The function D with the description D' is considered that predicts using S the result of the call $D(D')$. After that the computation follows the path consisting of the call of B if the endless computation is predicted and the path with the call of C — otherwise. In both cases the behaviour of either path and of the whole function contradicts to the prediction. Thus from the assumption made on the function S property the identically false result

$$\neg(S(D,\,D') = T)\ \wedge \neg(S(D,\,D') \neq T)$$

may be derived what leads to the conclusion that the function S with the required property cannot exist.

This proof may be considered perfect if it were possible to describe the function D which makes exactly all said above in connection with it. Evidently the description should look like this:

$$D(X) = \text{if } S(D,\,X) \text{ then } C() \text{ else } B()$$

What is the value of the expression $S(D,\,X)$ occuring within it? Acting straightforwardly one may try to replace this expression by $D(X)$ (since the latter may have only T as its value). However this trial leads to no result at all, since such a call of D sticks already within the condition of the rightpart of the function definition and neither of the branches may be chosen. May the roundabout ways help?

4 The Static vs. the Dynamic Approach

While analysing the function S behaviour the *static* approach takes into account only the text of the function A description so the value of S may be got either independently of the value of X or with very weak assumptions on the argument properties. The *dynamic* approach implies that the properties of current data are looked through the whole process of $A(X)$ evaluation and therefore the specific value of X should be given.

The assumption that a function property may be always revealed statically should be declined since just this approach leads to the universally false assertion. On the other hand the endless recursion bounded with the dynamic evaluation of $S(D,\,D')$ ruins the plan of getting a result with properties opposite to predicted ones.

As regards to the pair (D, D') the title question of section **3** may not be stated in a manner leading to any answer at all (the barber is bearded and the question — whether he shaves himself or not — looses any sense). Thus it is proved that existence of the function S may not be refuted using the function D.

So the diagonal method of reducing to a contradiction being so tempting in theory has not worked in practice. The discord between two colleagues in connection with the same method is coming to mind. However in books and papers it remains the leading method to prove assertions that algorithms of solving many problems are impossible. In that case such problems are called ***algorithmically unsolvable***.

The guile of the intention — to compose the function D so that it behaves in spite of the made assumption — is not tightly bound with the situation. The places where B and C are called may be interchanged to make D to behave in accordance with the prediction. However that does not make the prediction possible. Any function calling itself to bring a judgement whether it has some definite property is not shielded from endless looping.

5 A More General Case

Let the algorithms either having or not having a property P do both exist and a function similar to D inherits the property from the chosen branch. The so called Rice (or Uspensky-Rice, cf., e. g., [1, §56]) theorem states that no algorithm recognizing such a property is possible. In its proof the D-like function is used. The proof is as vulnerable as the previous one and on the same reason — the looping arises inevitably before any prediction is made. This vulnerability lies on the surface being detected statically.

Only very seldom and for a very simple algorithms their properties may be found without their execution. To determine that the result of an algorithm execution is bound in a certain manner with the input data is usually possible only while looking through and analysing the algorithm action with a certain variant of the data. Only in just such a formulation the assertion on algorithmic unsolvability nearly all mass problems of the computability theory may be accepted.

6 Selfapplicability

In the traditional theory the first place among these problems occupies that of selfapplicability of functions (cf. [1, §§46 and 47]). A function is called either ***selfapplicable*** or ***unselfapplicable*** depending on its applicability to its own description. The problem is stated: to build a function D, which is applicable only to the descriptions of all unselfapplicable functions (a variant of Russel's paradox). The impossibility of the building is proved by the same diagonal method.

The assumption that D is selfapplicable i. e. applicable to its description D' would mean in view of requirement to D that D' describes an unselfapplicable function. The arisen contradiction leads to the conclusion that D is unselfapplicable. When this is actually so no contradiction may arise. Indeed the unselfapplicability of D means that one should infinitely long wait for the result of application of D to D', i. e. the second outcome of the prediction never will be available. In the same way the Russel's barber never meets the question — to shave or not to shave himself as a client.

7 The Freedom as a Realized Necessity

In [1, the remark to § 47.2.1] the nonadmittance of a too large freedom in the context of set-theoretical conception is noted — it is impossible without falling in contradictions to combine freely any 'objects' in 'sets' which in their turn will be treated as 'objects'. However, on some reason there never arises the question whether without any limitation one may build 'words' — the descriptions of the 'algorithms' and to transfer these words on input of any algorithm.

Maybe in the context of algorithm-theoretical approach the freedom may turn to be superfluous too? Let e. g. a normal algorithm is written assuming that its input word consists of two parts with a delimiter between them. Is there any reason to allow its application to the word containing no such delimiter? In other words — it is not reasonable to violate the simplest and well known to programmers limitation on types and to call a function of two parameters with only one argument.

8 Conclusion

In this paper written on the minimal level of formalization the next assertions are grounded:

- just this level is only appropriate to the analysis of the problems taken from the so called foundations of mathematics;
- some of these problems need to take into account their researcher's influence on their results, especially in the case of the application of an algorithm to its own description;
- the contradiction grounding the impossibility of some algorithms arises only with the static approach, the dynamic one leads only to the impossibility to judge definitely on these algorithms behaviour;
- some limitations generally accepted in set theory and in programming should be observed in computability theory as well, their violation leads to the unpredictable consequences.

References

1. A. A. Markov, N. M. Nagorny. Algorithms theory — Moscow, 1996 (in Russian)
2. E. Mendelson. Introduction to mathematical logic — Princeton etc.: van Nostrand Company, inc. (Russian translation: Moscow, 1976)

Resolution and Binary Decision Diagrams Cannot Simulate Each Other Polynomially

Jan Friso Groote and Hans Zantema

Department of Computer Science, Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: jfg@win.tue.nl, h.zantema@tue.nl

Abstract. There are many different ways of proving formulas in propositional logic. Many of these can easily be characterized as forms of resolution (e.g. [13] and [10]). Others use so-called binary decision diagrams (BDDs) [2, 11]. Experimental evidence suggests that BDDs and resolution based techniques are fundamentally different, in the sense that their performance can differ very much on benchmarks [15]. In this paper we confirm these findings by mathematical proof. We provide examples that are easy for BDDs and exponentially hard for any form of resolution, and vice versa, examples that are easy for resolution and exponentially hard for BDDs.

1 Introduction

We consider formulas in proposition logic: formulas consisting of proposition letters from some set $\mathcal{P}$, constants t (true) and f (false) and connectives $\vee$, $\wedge$, $\neg$, $\rightarrow$ and $\leftrightarrow$. There are different ways of proving the correctness of these formulas, i.e., proving that a given formula is a tautology. In the automated reasoning community resolution is a popular proof technique, underlying the vast majority of all proof search techniques in this area, including for instance the well known branch-and-bound based technique named after Davis-Putnam-Loveland [6] or the remarkably effective methods by Stålmarck [13] and the GRASP prover [10].

In the VLSI and the process analysis communities binary decision diagrams (BDDs) are popular [2, 11]. BDDs have caused a considerable increase of the scale of systems that can be verified, far beyond anything a resolution based method has achieved. On the other hand there are many examples where resolution based techniques out-perform BDDs with a major factor, for instance in proving safety of railway interlockings ([8]). Out-performance in both directions has been described in [15].

However, benchmark studies only provide an impression, saying very little about the real relation of resolution and BDDs. The results may be influenced by badly chosen variable orderings in BDDs or non optimal proof search strategies in resolution. Actually, given such benchmarks it can not be excluded that there exist a resolution based technique that always out-performs BDDs, provided a proper proof search strategy would be chosen. So, a mathematical comparison between the techniques is called for. This is not straightforward, as resolution and BDDs look very different. BDDs work on arbitrary formulas, whereas resolution is strictly linked to formulas in conjunctive normal form. And the resolution rule and the BDD construction algorithms appear of a totally dissimilar nature.

Moreover, classical (polynomial) complexity bounds cannot be used, as the problem we are dealing with is (co-)NP-complete. Fortunately, polynomial simulations provide an elegant way of dealing with this (see e.g. [17]). We say that proof system A polynomially simulates proof system B if for every formula ϕ the size of the proof of ϕ in system A is smaller than a polynomial applied to the size of the proof of ϕ in system B. Of course, if the polynomial is more than linear, proofs in system A may still be substantially longer than proofs in system B, but at least the proofs in A are never exponentially longer. It is self evident that for practical applications it is important that the order of the polynomial is low. If it can be shown that for some formulas in B the proofs are exponentially longer than those in A we consider A as a strictly better proof system than B. It has for instance been shown that 'extended resolution' is strictly better than resolution [9], being strictly better than Davis-Putnam resolution [7]; for an extended overview of comparisons of systems based on resolution, Frege systems and Gentzen systems we refer to [17].

We explicitly construct a sequence of biconditional formulas that are easy for BDDs, but exponentially hard for resolution. The proof that they are indeed hard for resolution is based on results from [16, 1].

The reverse is easier, namely showing that there is a class of formulas easy for any reasonable form of resolution, even only unit resolution, and exponentially hard for BDDs. For a suitable class of formulas including pigeon hole formulas we prove that the BDD approach is exponentially hard. It was proven before in [9] that for the same pigeon hole formulas resolution is exponentially hard for every strategy.

Both directions of this main result we prove by giving an explicit simple construction for a sequence of formulas for which the gap between both methods is proved. For both directions a non-constructive counting argument that such a sequence of formulas exists would be simpler, but we prefer the constructive approach.

We start with preliminaries on OBDDs in Section 2. In Section 3 we prove that OBDD proofs are exponential for pigeon hole formulas and related formulas. In Section 4 we prove that OBDD proofs are polynomial for biconditional formulas. In Section 5 we present our results on resolution. In Section 6 we present our main results in comparing resolution and OBDDs. Finally, in Section 7 we describe some points of further research.

Acknowledgment. Special thanks go to Oliver Kullmann and Alasdair Urquhart for their help with lower bounds for resolution.

2 Binary Decision Diagrams

The kind of Binary Decision Diagrams that we use presupposes a total ordering $<$ on $\mathcal{P}$, and therefore are also called Ordered Binary Decision Diagrams (OBDDs). First we present some basic definitions and properties as they are found in e.g. [2, 11]. An OBDD is a Directed Acyclic Graph (DAG) where each node is labeled by a proposition letter from $\mathcal{P}$, except for nodes that are labeled by 0 and 1. From every node labeled by a proposition letter, there are two outgoing edges, labeled 'left' and 'right', to nodes labeled by 0 or 1, or a proposition letter strictly higher in the ordering $>$. The nodes labeled by 0 and 1 do not have outgoing edges.

An OBDD compactly represents which valuations are valid, and which are not. Given a valuation σ and an OBDD B, the σ walk of B is determined by starting at the root of the DAG, and iteratively following the left edge if σ validates the label of the current node, and otherwise taking the right edge. If 0 is reached by a σ-walk then B makes σ invalid, and if 1 is reached then B makes σ valid. We say that an OBDD represents a formula if the formula and the OBDD validate exactly the same valuations.

An OBDD is called *reduced* if the following two requirements are satisfied.

1. For no node do its left and right edge go to the same node. It is straightforward to see that a node with such a property can be removed. We call this the *eliminate* operation.
2. There are no two nodes with the same label of which the left edges go to the same node, and the right edges go to the same node. If this is the case these nodes can be taken together, which we call the *merge* operation.

Applying the merge and the eliminate operator to obtain a reduced OBDD can be done in linear time. Reduced OBDDs have the following very nice property.

Lemma 1 *For a fixed order $<$ on $\mathcal{P}$, every propositional formula ϕ is uniquely represented by a reduced OBDD $B(\phi,<)$, and ϕ and ψ are equivalent if and only if $B(\phi,<) = B(\psi,<)$.*

As a consequence, a propositional formula ϕ is a contradiction if and only if $B(\phi,<) = 0$, and it is a tautology if and only if $B(\phi,<) = 1$. Hence by computing $B(\phi,<)$ for any suitable order $<$ we can establish whether ϕ is a contradiction, or ϕ is a tautology, or ϕ is satisfiable. If the order $<$ is fixed we shortly write $B(\phi)$ instead of $B(\phi,<)$. We write $\#(B(\phi))$ for the number of internal nodes in $B(\phi)$.

The main ingredient for the computation of $B(\phi)$ is the *apply*-operation: given the reduced OBBDs $B(\phi)$ and $B(\psi)$ for formulas ϕ and ψ and a binary connective $\diamond \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$ as parameters, the *apply*-operation computes $B(\phi \diamond \psi)$. For the usual implementation of *apply* as described in [2, 11] both time and space complexity are $O(\#(B(\phi)) * \#(B(\psi)))$. If $B(\phi)$ is known then $B(\neg\phi)$ is computed in linear time simply by replacing every 0 by 1 and vice versa; this computation is considered as a particular case of an *apply*-operation. Now for every

ϕ its reduced OBDD $B(\phi)$ can be computed by recursively calling the *apply*-operation. As the basis of this recursion we need the reduced OBDDs for the single proposition letters. These are simple: the reduced OBDD for p consists of a node labeled by p, having a left outgoing edge to 0 and a right outgoing edge to 1. By maintaining a hash-table for all sub-formulas it can be avoided that for multiple occurrences of sub-formulas the reduced OBDD is computed more than once.

By the *OBDD proof* of a formula ϕ we mean the recursive computation of $B(\phi)$ using the *apply*-operation as described above. If ϕ consists of n boolean connectives then this proof consists of exactly n calls of the *apply*-operation. However, by the expansion of sizes of the arguments of *apply* this computation can be of exponential complexity, even if it ends in $B(\phi) = 0$. As the satisfiability problem is NP-complete, this is expected to be unavoidable for every way to compute $B(\phi)$. We give an explicit construction of formulas for which we prove that the resulting OBDDs and hence the OBDD proofs are of exponential size, independently of the order $<$ on $\mathcal{P}$. In our main result this is applied by the observation that the OBDD proof of $p \wedge (\neg p \wedge \phi)$ is long for such a formula ϕ for which $B(\phi)$ is large, while the resolution proof is short.

In [3] it was proved that representing the middle bits of a binary multiplier requires an exponential OBDD; this function is easily represented by a small circuit, but not by a small formula, and hence does not serve for our goal of having a small formula with an exponential OBDD proof.

3 Pigeon Hole Formulas

In this section we prove lower bounds for OBDD proofs for pigeon hole formulas and related formulas.

Definition 2 *Let m, n be positive integers and let p_{ij} be distinct variables for $i = 1, \ldots, m$ and $j = 1, \ldots, n$. Let*

$$C_{m,n} = \bigwedge_{i=1}^{m}(\bigvee_{j=1}^{n} p_{ij}), \qquad R_{m,n} = \bigwedge_{j=1}^{n}(\bigvee_{i=1}^{m} p_{ij}), \qquad \overline{R}_{m,n} = \bigwedge_{j=1,\ldots,n,1\leq i<k\leq m} (\neg p_{ij} \vee \neg p_{kj}),$$

$$CR_{m,n} = C_{m,n} \wedge R_{m,n}, \qquad PF_{m,n} = C_{m,n} \wedge \overline{R}_{m,n}.$$

In order to understand these formulas put the variables in a matrix according to the indexes. The formula $C_{m,n}$ states that in every of the m columns at least one variable is true, the formula $R_{m,n}$ states that in every of the n rows at least one variable is true, and the formula $\overline{R}_{m,n}$ states that in every of the n rows at most one variable is true. Hence if $C_{m,n}$ holds then at least m of the variables p_{ij} are true and if $\overline{R}_{m,n}$ holds then at most n of the variables p_{ij} are true. Hence if $m > n$ then $PF_{m,n}$ is a contradiction. Since this reasoning describes the well-known pigeon hole principle, the formulas $PF_{m,n}$ are called pigeon hole formulas. Note that $PF_{m,n}$ is in conjunctive normal form. In [9] it has been proved that for every resolution proof for $PF_{n+1,n}$ the length is at least exponential in n. Here we prove a similar exponential lower bound for OBDD proofs, which is of interest in itself since pigeon hole formulas are widely considered as benchmark formulas. For the main result of the paper however we get better results by using similar lower bounds for $CR_{m,n}$ instead since the size of $CR_{n,n}$ is quadratic in n while pigeon hole formulas have cubic sizes. The contradictory formula in the main result is $p \wedge (\neg p \wedge CR_{n,n})$.

Our proof of these lower bounds has been inspired by the proof from [15] that every OBDD for $CR_{n,n}$ has a size that is exponential in $\sqrt{n}$, which we improve to a size that is exponential in n. First we need two lemmas.

Lemma 3 *Let ϕ be a formula over variables in any finite set $\mathcal{P}$. Let $<$ be a total order on $\mathcal{P}$. Let $k < \#\mathcal{P}$. Write $\mathbb{B} = \{0, 1\}$. Let $f_\phi : \mathbb{B}^{\#\mathcal{P}} \rightarrow \mathbb{B}$ the function representing ϕ, in such a way that the smallest k elements of $\mathcal{P}$ with respect to $<$ correspond to the first k arguments of f_ϕ. Let $A \subseteq \{1, \ldots, k\}$. Let $z \in \mathbb{B}^k$. Assume that for every distinct $x, x' \in \mathbb{B}^k$ satisfying $x_i = x'_i = z_i$ for all $i \notin A$ there exists $y \in \mathbb{B}^{\#\mathcal{P}-k}$ such that $f_\phi(x, y) \neq f_\phi(x', y)$. Then $\#B(\phi, <) \geq 2^{\#A}$.*

Proof: There are $2^{\#A}$ different ways to choose $x \in \mathbb{B}^k$ satisfying $x_i = z_i$ for all $i \notin A$. Now from the assumption it is clear that by fixing the first k arguments of f_ϕ, at least $2^{\#A}$ different functions in the remaining $\#\mathcal{P} - k$ arguments are obtained. All of these functions correspond to different nodes in the reduced OBDD $B(\phi, <)$, proving the lemma. □

Lemma 4 *Let $m, n \geq 1$. Consider a matrix of n rows and m columns. Let the matrix entries be colored equally white and black, i.e., the difference between the number of white entries and the number of black entries is at most one. Then at least $\frac{(m-1)\sqrt{2}}{2}$ columns or at least $\frac{(n-1)\sqrt{2}}{2}$ rows contain both a black and a white entry.*

Proof: If all rows contain both a black and a white entry we are done, so we may assume that at least one row consists of entries of the same color. By symmetry we may assume all entries of this row are white. If also a row exists with only black entries, then all columns contain both a black and a white entry and we are done. Since there is a full white row, we conclude that no full black column exists. Let r be the number of full white rows and c be the number of full white columns. The number of entries in these full white rows and columns together is $mr + cn - cr$, and the total number of white entries is at most $\frac{mn+1}{2}$, hence

$$\frac{mn+1}{2} \geq mr + cn - cr = mn - (m-c)(n-r).$$

Assume the lemma does not hold. Then $m - c < \frac{(m-1)\sqrt{2}}{2}$ and $n - r < \frac{(n-1)\sqrt{2}}{2}$, and

$$\frac{mn+1}{2} \geq mn - (m-c)(n-r) > mn - \frac{(m-1)\sqrt{2}}{2} * \frac{(n-1)\sqrt{2}}{2} = mn - \frac{(m-1)(n-1)}{2}$$

from which we conclude $m + n < 2$, contradiction. □

Theorem 5 *For $m \geq n \geq 1$ and for every total order $<$ on $\mathcal{P} = \{p_{ij} | i = 1, \ldots, m, j = 1, \ldots, n\}$ both time and space complexity of the OBDD proofs of both $CR_{m,n}$ and $PF_{m,n}$ is $\Omega(1.63^n)$. Moreover, $\#B(CR_{m,n}, <) = \Omega(1.63^n)$.*

Proof: The last step in the OBDD proof of $PF_{m,n}$ is the application of *apply* on $B(C_{m,n}, <)$ and $B(\overline{R}_{m,n}, <)$.

We prove that $\#B(CR_{m,n}, <) \geq 2^{\frac{(n-1)\sqrt{2}}{2}}$ and that either $B(C_{m,n}, <)$ has size at least $2^{\frac{(m-1)\sqrt{2}}{2}}$ or $B(\overline{R}_{m,n}, <)$ has size at least $2^{\frac{(n-1)\sqrt{2}}{2}}$. Since $m \geq n$ and $2^{\frac{\sqrt{2}}{2}} > 1.63$, then the theorem immediately follows.

Let $\mathcal{P}_< \subset \mathcal{P}$ consist of the $\lfloor \frac{nm}{2} \rfloor$ smallest elements of $\mathcal{P}$ with respect to $<$, and let $\mathcal{P}_> = \mathcal{P} \setminus \mathcal{P}_<$. Hence elements of $\mathcal{P}_>$ are greater than elements of $\mathcal{P}_<$. We say that row $j = \{p_{ij} | i = 1, \ldots, m\}$ is mixed if i, i' exist such that $p_{ij} \in \mathcal{P}_<$ and $p_{i'j} \in \mathcal{P}_>$; we say that column $i = \{p_{ij} | j = 1, \ldots, n\}$ is mixed if j, j' exist such that $p_{ij} \in \mathcal{P}_<$ and $p_{ij'} \in \mathcal{P}_>$.

From Lemma 4 we conclude that either at least $\frac{(n-1)\sqrt{2}}{2}$ rows are mixed or at least $\frac{(m-1)\sqrt{2}}{2}$ columns are mixed. For both cases we will apply Lemma 3 for $k = \lfloor \frac{nm}{2} \rfloor$. We number the elements of $\mathcal{P}$ from 1 to mn such that the numbers $1, \ldots, k$ correspond to the elements of $\mathcal{P}_<$.

Assume that at least $\frac{(m-1)\sqrt{2}}{2}$ columns are mixed. In case all columns are mixed, separate one of them and consider it to be non-mixed. For every mixed column fix one element of $\mathcal{P}_<$ in that column; collect the numbers of these elements in the set A. For $i = 1, \ldots, k$ define $z_i = 1$ for i corresponding to matrix elements in non-mixed columns and $z_i = 0$ for i corresponding to matrix elements in mixed columns. Choose $\boldsymbol{x}, \boldsymbol{x}' \in \mathbb{B}^k$ satisfying $\boldsymbol{x} \neq \boldsymbol{x}'$ and $x_i = x'_i = z_i$ for all $i \notin A$. Then there exists $i \in A$ such that $x_i \neq x'_i$. Now let $\boldsymbol{y} = (y_{k+1}, \ldots, y_{mn})$ be the vector defined by $y_j = 0$ if $j \in \mathcal{P}_>$ corresponds to a matrix element in the same column as i, and $y_j = 1$ otherwise. Interpret the concatenation of $\boldsymbol{x}$ and $\boldsymbol{y}$ as an assignment to $\{0, 1\}$ on the matrix entries. Non-mixed columns contain only the value 1, and every mixed column contains at least one value 1, except for one column which consists purely of zeros if and only if $x_i = 0$. Since we forced at least one column to be considered as non-mixed and containing only the value 1, every row contains at least one value 1. Hence $f_{CR_{m,n}}(\boldsymbol{x}, \boldsymbol{y}) = f_{C_{m,n}}(\boldsymbol{x}, \boldsymbol{y}) = x_i$, and similarly $f_{CR_{m,n}}(\boldsymbol{x}', \boldsymbol{y}) = f_{C_{m,n}}(\boldsymbol{x}', \boldsymbol{y}) = x'_i$. Since $x_i \neq x'_i$ we obtain $f_{C_{m,n}}(\boldsymbol{x}, \boldsymbol{y}) \neq f_{C_{m,n}}(\boldsymbol{x}', \boldsymbol{y})$ and $f_{CR_{m,n}}(\boldsymbol{x}, \boldsymbol{y}) \neq f_{CR_{m,n}}(\boldsymbol{x}', \boldsymbol{y})$. Now by Lemma 3 we conclude that $\#B(C_{m,n}, <) \geq 2^{\#A} \geq 2^{\frac{(m-1)\sqrt{2}}{2}}$ and $\#B(CR_{m,n}, <) \geq 2^{\#A} \geq 2^{\frac{(m-1)\sqrt{2}}{2}} \geq 2^{\frac{(n-1)\sqrt{2}}{2}}$.

For the remaining case assume that at least $\frac{(n-1)\sqrt{2}}{2}$ rows are mixed. The required bound for $\#B(CR_{m,n}, <)$ follows exactly as above by symmetry. It remains to prove the bound for $\#B(\overline{R}_{m,n}, <)$. For every mixed row fix one element of $\mathcal{P}_<$ in that row; collect all these elements in the set A. Define $z_i = 0$ for all $i = 1, \ldots, k$. Choose $\boldsymbol{x}, \boldsymbol{x}' \in \mathbb{B}^k$ satisfying $\boldsymbol{x} \neq \boldsymbol{x}'$ and $x_i = x'_i = z_i = 0$ for all $i \notin A$. Then there exists $i \in A$ such that $x_i \neq x'_i$. Now define $\boldsymbol{y} = (y_{k+1}, \ldots, y_{mn})$ by choosing $y_j = 0$ for all but one j, and $y_j = 1$ for one single j for which i and j correspond to matrix elements in the same row. This is possible because i corresponds to an entry in a mixed row. Since in every other row at most one value is set to 1 all corresponding clauses in $\overline{R}_{m,n}$ are true. The only clause in $\overline{R}_{m,n}$ that is possibly false is the one corresponding to i and j. We obtain $f_{\overline{R}_{m,n}}(\boldsymbol{x}, \boldsymbol{y}) = \neg x_i$ and $f_{\overline{R}_{m,n}}(\boldsymbol{x}', \boldsymbol{y}) = \neg x'_i$. Since $x_i \neq x'_i$ we have $f_{\overline{R}_{m,n}}(\boldsymbol{x}, \boldsymbol{y}) \neq f_{\overline{R}_{m,n}}(\boldsymbol{x}', \boldsymbol{y})$. Now by Lemma 3 we conclude that $\#B(\overline{R}_{m,n}, <) \geq 2^{\#A} \geq 2^{\frac{(n-1)\sqrt{2}}{2}}$. □

Note that we proved that either $C_{m,n}$ or $\overline{R}_{m,n}$ must have an OBDD of exponential size. However, for each of these formulas separately a properly chosen order may lead to small OBDDs. Indeed, if

$$p_{ij} < p_{i'j'} \iff (i < i') \vee (i = i' \wedge j < j')$$

then $\#B(C_{m,n},<) = mn$ and if

$$p_{ij} < p_{i'j'} \iff (j < j') \vee (j = j' \wedge i < i')$$

then $\#B(R_{m,n},<) = mn$ and $\#B(\overline{R}_{m,n},<) = 2(m-1)n$, all being linear in the number of variables.

4 Biconditional Formulas

An interesting class of formulas are *biconditional formulas* consisting of proposition letters, biconditionals ($\leftrightarrow$) and negations ($\neg$). Biconditionals have very nice properties: they are associative, $\phi \leftrightarrow (\psi \leftrightarrow \chi) \equiv (\phi \leftrightarrow \psi) \leftrightarrow \chi$, commutative, $\phi \leftrightarrow \psi \equiv \psi \leftrightarrow \phi$, idempotent, $\phi \leftrightarrow \phi \equiv \mathrm{t}$ and satisfy $\phi \leftrightarrow \neg\psi \equiv \neg(\phi \leftrightarrow \psi)$.

For a string $S = p_1, p_2, p_3, \ldots, p_n$ of proposition letters, where letters are allowed to occur more than once, we write

$$[S] \ = \ p_1 \leftrightarrow (p_2 \leftrightarrow (p_3 \cdots (p_{n-1} \leftrightarrow p_n)) \cdots).$$

It is not difficult to see that $[S]$ is a tautology if and only if all letters occur an even number of times in S.

A formula of the shape $[S]$ or $\neg[S]$ for a string S in which every symbol occurs at most once, is called a *biconditional normal form.* Using the above properties it is easy to show that for every biconditional formula there exists a logically equivalent biconditional normal form.

The BDD technique turns out to be very effective for biconditional formulas. We show that for any biconditional formula its OBDD proof has a polynomial complexity. For any biconditional formula ϕ, we write $|\phi|$ for the size of ϕ, $\alpha(\phi)$ for the number of variables occurring in ϕ and $\alpha_{odd}(\phi)$ for the number of variables that occur an odd number of times in ϕ.

It is useful to speak about the OBDD of n formulas, $\phi_1, \ldots, \phi_n$. This OBDD is a single DAG with up to n root nodes. The notion *reduced* carries over to these OBDDs. In particular, if ϕ_i and ϕ_j are equivalent, then the i^{th} and j^{th} root node are the same. Again the size of a DAG is defined to be the number of its internal nodes.

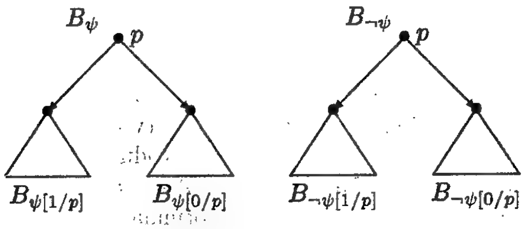
We have the following lemma, showing that each reduced OBDD for a biconditional formula is small.

Lemma 6 *Let ϕ be a biconditional formula. Any reduced OBDD for ϕ and $\neg\phi$ has size $2\alpha_{odd}(\phi)$.*

Proof: First fix an arbitrary ordering $<$ on the proposition letters. Note that there is a biconditional normal form ψ that is equivalent to ϕ. As by Lemma 1 the reduced OBDD of ϕ and ψ are the same, we can as well construct the OBDD of ψ. Moreover, $\alpha_{odd}(\phi) = \alpha_{odd}(\psi)$.

We prove the lemma by induction on $\alpha_{odd}(\psi)$.

- $\alpha_{odd}(\psi) = 0$. As ψ is a biconditional normal form, it does not contain any proposition letter, and hence is either equivalent to true or false. So, the reduced OBDD of ϕ and $\neg\phi$ does not contain internal nodes at all, and has size 0.
- $\alpha(\psi)_{odd} = n + 1$. Consider the first letter in the ordering $<$ that occurs in ψ and let it be p. The OBDDs for ψ and $\neg\psi$ look like:



Here $\psi[v/p]$ is the formula ψ where v has been substituted for p. Clearly, as p occurs an odd time in ψ, $\psi[0/p] \equiv \neg\psi[1/p]$ and $\psi[1/p] \equiv \neg\psi[0/p]$. So, the reduced OBDD of $\psi[0/p]$, $\neg\psi[1/p]$, $\psi[1/p]$ and $\neg\psi[0/p]$ is the same as the OBDD of $\psi[0/p]$ and $\neg\psi[0/p]$. Using the induction hypothesis, the size of this OBDD must be $2n$. The reduced OBDD for ψ and $\neg\psi$ adds two new nodes. So, the size of the reduced OBDD of ψ and $\neg\psi$ is $2n+2$. This equals $2\alpha_{odd}(\psi) + 2$, finishing the proof. □

Theorem 7 *Let $<$ be an ordering on the proposition letters.*

- *The complexity of the corresponding OBDD proof for any biconditional formula ϕ is $O(|\phi|^3)$.*
- *The complexity of the corresponding OBDD proof for $[S]$ or $\neg[S]$ for any string S of proposition letters is $O(|[S]|^2)$.*

Proof: The OBDD proof for ϕ consists of $O(|\phi|)$ applications of *apply* applied on reduced OBDDs of sub-formulas of ϕ. By Lemma 6 each of these reduced OBDDs has size $O(|\phi|)$. Since the complexity of *apply*$(\leftrightarrow, B, B')$ is $O(\#B * \#B')$ and the complexity of *apply*$(\neg, B)$ is $O(\#B)$ for every *apply* operation the complexity is $O(|\phi|^2)$, yielding $O(|\phi|^3)$ for the full OBDD proof for ϕ.

For the OBDD proof for $[S]$ or $\neg[S]$ only applications of *apply*$(\leftrightarrow, B, B')$ occur with $\#B = 1$, giving the complexity $O(\#B')$, yielding $O(|[S]|^2)$ for the full OBDD proof. □

5 Resolution

Resolution is a very common technique to prove formulas. Contrary to the BDD technique, it is applied to formulas in *conjunctive normal form* (*CNF*), i.e. formulas of the form

$$\bigwedge_{i\in I}\bigvee_{j\in J_i} l_{ij}$$

where I and J_i are finite index sets and l_{ij} is a literal, i.e. a formula of the form p or $\neg p$ for a proposition letter p. Each sub-formula $\bigvee_{j\in J_i} l_{ij}$ is called a *clause*. As $\wedge$ and $\vee$ are associative, commutative and idempotent it is allowed and convenient to view clauses as sets of literals and CNFs as sets of clauses.

The resolution rule can be formulated by:

$$\frac{\{p, l_1, \ldots, l_n\} \quad \{\neg p, l'_1, \ldots, l'_{n'}\}}{\{l_1, \ldots, l_n, l'_1, \ldots, l'_{n'}\}}$$

where p is a proposition letter and l_i, l'_j are literals. A resolution proof of a set of clauses F is a sequence of clauses where the last clause is empty and each clause in the sequence is either taken from F, or matches the conclusion of the resolution rule, where both premises occur earlier in the sequence. Such a resolution sequence ending in the empty clause is called a *resolution refutation*, and proves that the conjunction of the set of clauses is a contradiction.

In case one of the clauses involved is a single literal l, by this resolution rule all occurrences of the negation of l in all other clauses may be removed. Moreover, all other clauses containing l then may be ignored. Eliminating all occurrences of l and its negation in this way is called *unit resolution*. All practical resolution proof search systems start with doing unit resolution as long as possible.

In order to apply resolution on arbitrary formulas, these formulas must first be translated to CNF. This can be done in linear time maintaining satisfiability using the Tseitin transformation [14]. A disadvantage of this transformation is the introduction of new variables, but it is well-known that a transformation to CNF without the introduction of new variables is necessarily exponential. For instance, it is not difficult to prove that for

$$(\cdots((p_1 \leftrightarrow p_2) \leftrightarrow p_3) \cdots\cdots \leftrightarrow p_n)$$

every clause in a CNF contains either p_i or $\neg p_i$ for every i. Since one such clause of n literals causes only one zero in the truth table of the formula, the full CNF contains 2^{n-1} of these clauses to obtain all 2^{n-1} zeros in its truth table. Hence without the introduction of new variables every CNF of this formula is of exponential size. More general for every biconditional formula ϕ without the introduction of new variables every CNF consists of at least $2^{\alpha_{odd}(\phi)-1}$ clauses each consisting of at least $\alpha_{odd}(\phi)$ literals.

The *Tseitin transformation* works as follows. Given a formula ϕ. Every sub-formula ψ of ϕ not being a proposition letter is assigned a new letter p_ψ. Now the Tseitin transformation of ϕ consists of

- the single literal p_ϕ;
- the conjunctive normal form of $p_\psi \leftrightarrow (p_{\psi_1} \diamond p_{\psi_2})$ for every subterm ψ of ϕ of the shape $\psi = \psi_1 \diamond \psi_2$ for a binary operator $\diamond$;
- the conjunctive normal form of $p_\psi \leftrightarrow \neg p_{\psi_1}$ for every subterm ψ of ϕ of the shape $\psi = \neg\psi_1$.

Here p_{ψ_i} is identified with ψ_i in case ψ_i is a proposition letter, for $i = 1, 2$. It is easy to see that this set of clauses is satisfiable if and only if ϕ is satisfiable. Moreover, every clause consists of at most three literals, and the number of clauses is linear in the size of the original formula ϕ.

It is not difficult to see that after applying the Tseitin transformation to a CNF, by a number of resolution steps linear in the size of the CNF, the original CNF can be re-obtained. By a resolution proof for an arbitrary formula we mean a resolution proof after the Tseitin transformation has been applied.

We now give a construction of strings S_n in which all letters occur exactly twice by which $\neg[S_n]$ is a contradiction, and for which we prove that every resolution proof of $\neg[S_n]$ is very long. The crucial idea is that the Tseitin transformation applied to this formula $\neg[S_n]$ coincides with the Tseitin contradiction of a graph of high expansion, using the terminology of [1].

Although the construction is somewhat involved, we think that simpler constructions do not suffice. In [17] for instance it was proved that $\neg[p_1, p_2, \ldots, p_n, p_1, p_2, \ldots, p_n]$ admits a resolution proof that is quadratic in n. A slightly better construction can be given based on expander graphs, but giving such a construction constructively is much more complicated than ours while the difference is only a logarithmic factor.

For a string S and a label i we write $\mathsf{lab}(S, i)$ for the string obtained from S by replacing every symbol p by a fresh symbol p_i. For a string S of length $n * 2^n$ we write $\mathsf{ins}(n, S)$ for the string obtained from S by inserting the symbol i after the $(i * n)$-th symbol for $i = 1, 2, \ldots, n$. We define

$$S_1 = 1, 1, \quad \text{and}$$

$$S_{n+1} = \mathsf{ins}(n, \mathsf{lab}(S_n, 0)), \mathsf{ins}(n, \mathsf{lab}(S_n, 1)),$$

for $n > 0$. For instance, we have

$$S_1 = \underbrace{1}, \underbrace{1},$$

$$S_2 = \underbrace{1_0, 1}, \underbrace{1_0, 2}, \underbrace{1_1, 1}, \underbrace{1_1, 2},$$

$$S_3 = \underbrace{1_{00}, 1_0, 1}, \underbrace{1_{00}, 2_0, 2}, \underbrace{1_{10}, 1_0, 3}, \underbrace{1_{10}, 2_0, 4}, \underbrace{1_{01}, 1_1, 1}, \underbrace{1_{01}, 2_1, 2}, \underbrace{1_{11}, 1_1, 3}, \underbrace{1_{11}, 2_1, 4}.$$

Clearly S_n is a string of length $n * 2^n$ over $n * 2^{n-1}$ symbols each occurring exactly twice. The string S_n can be considered to consist of 2^n consecutive groups of n symbols, called n-groups. In the examples S_1, S_2 and S_3 above the n-groups are under-braced. Write $g_{n,k}$ to be the k-th n-group in S_n, for $n > 1$ and $1 \leq k \leq 2^n$.

Lemma 8 *Let $A \subseteq \{1, 2, \ldots, 2^n\}$ for any $n > 0$. Then there are at least $\min(\#A, 2^n - \#A)$ pairs (k, k') such that $k, k' \in \{1, 2, \ldots, 2^n\}$, $k \in A$, $k' \notin A$ and $g_{n,k}$ and $g_{n,k'}$ have a common symbol.*

Proof: We apply induction on n; for $n = 1$ the lemma clearly holds. Let $m_0 = \#\{k \in A | k \leq 2^{n-1}\}$ and $m_1 = \#\{k \in A | k > 2^{n-1}\}$. Say that (k, k') is a matching pair if $k \in A$, $k' \notin A$ and $g_{n,k}$ and $g_{n,k'}$ have a common symbol. If $k, k' \leq 2^{n-1}$ then by construction $g_{n,k}$ and $g_{n,k'}$ have a common symbol if $g_{n-1,k}$ and $g_{n-1,k'}$ have a common symbol. If $k, k' > 2^{n-1}$ then by construction $g_{n,k}$ and $g_{n,k'}$ have a common symbol if $g_{n-1,k-2^{n-1}}$ and $g_{n-1,k'-2^{n-1}}$ have a common symbol. Hence by induction hypothesis there are at least $\min(m_0, 2^{n-1} - m_0)$ matching pairs (k, k') with $k, k' \leq 2^{n-1}$ and at least $\min(m_1, 2^{n-1} - m_1)$ matching pairs (k, k') with $k, k' > 2^{n-1}$. Since by construction $g_{n,k}$ and $g_{n,k+2^{n-1}}$ have a common symbol for every $k = 1, 2, \ldots, 2^{n-1}$, there are at least $|m_0 - m_1|$ matching pairs (k, k') with $|k - k'| = 2^{n-1}$. Hence the total number of matching pairs is at least

$$|m_0 - m_1| + \min(m_0, 2^{n-1} - m_0) + \min(m_1, 2^{n-1} - m_1).$$

A simple case analysis shows that this is at least $\min(m_0 + m_1, 2^n - m_0 - m_1) = \min(\#A, 2^n - \#A)$. □

Essentially this lemma states the well-known fact that for any set A of vertices of an n-dimensional cube there are at least $\min(\#A, 2^n - \#A)$ edges for which one end is in A and the other is not. It is applied in the next lemma stating a lower bound on connections between separate elements of S_n rather than connections between n-groups.

Lemma 9 *Let $n > 0$ and let $B \subseteq \{1, 2, \ldots, n * 2^n\}$. Let $X \subseteq \{1, 2, \ldots, n * 2^n\}^2$ consist of the pairs (i, j) for which $i \in B$ and $j \notin B$ and for which either $|i - j| = 1$ or the i-th element of S_n is equal to the j-th element of S_n. Then*

$$\#X \geq \frac{\min(\#B, n * 2^n - \#B)}{2n}.$$

Proof: Assume that $\#B \leq n * 2^{n-1}$, otherwise replace B by its complement. Let A be the set of numbers $k \in \{1, \ldots, 2^n\}$ for which all elements of the corresponding n-group $g_{n,k}$ correspond to elements of B, i.e., $\{(k-1)*n+1, \ldots, k*n\} \subseteq B$. Let $m_1 = \#A$. Let m_2 be the number of n-groups for which none of the elements correspond to elements of B, i.e., $m_2 = \#\{k \in \{1, \ldots, 2^n\} | \{(k-1)*n+1, \ldots, k*n\} \cap B = \emptyset\}$. Let m_3 be the number of remaining n-groups, i.e., n-groups containing elements corresponding to both elements of B and outside B. Clearly $n * m_1 \leq \#B \leq n * (m_1 + m_3)$. Each of the m_3 remaining groups gives rise to a pair $(i, j) \in X$ for which $|i - j| = 1$. Hence $\#X \geq m_3$.

Now assume that $m_1 > m_3$. Since $n * m_1 \leq \#B \leq n * 2^{n-1}$ we have $m_1 = \#A \leq 2^{n-1}$. By Lemma 8 we obtain at least m_1 pairs (k, k') such that $k \in A$, $k' \notin A$ and $g_{n,k}$ and $g_{n,k'}$ have a common symbol. For at least $m_1 - m_3$ of the corresponding n-groups $g_{n,k'}$ none of the elements correspond to elements of B. Since $g_{n,k}$ and $g_{n,k'}$ have a common symbol for every corresponding pair (k, k') this gives rise to at least $m_1 - m_3$ pairs $(i, j) \in X$ for which the i-th element of S_n is equal to the j-th element of S_n. Hence in case $m_1 > m_3$ we conclude $\#X \geq m_3 + (m_1 - m_3) = m_1$.

We conclude

$$\#X \geq \max(m_3, m_1) \geq \frac{m_1 + m_3}{2} \geq \frac{\#B}{2n}.$$

□

Theorem 10 *Every resolution proof of $\neg[S_n]$ contains $2^{\Omega(2^n/n)}$ resolution steps.*

Proof: Let $S_n = p_1, p_2, \ldots, p_{n2^n}$; note that for every i there exists exactly one j with $p_i = p_j$ and $i \neq j$. Introduce distinct help symbols $q_0, q_1, q_2, \ldots, q_{n2^n-1}$. Now the Tseitin transformation of $\neg[S_n]$ consists of

- the single literal q_0;
- the conjunctive normal form of $q_0 \leftrightarrow \neg q_1$;
- the conjunctive normal form of $q_i \leftrightarrow (p_i \leftrightarrow q_{i+1})$ for every $i = 1, 2, \ldots, n * 2^n - 2$;
- the conjunctive normal form of $q_i \leftrightarrow (p_i \leftrightarrow p_{i+1})$ for $i = n * 2^n - 1$.

This set of clauses is exactly the same as $\tau(G, f)$, where τ is Tseitin's graph construction [14] also described in [16, 17, 1] for the graph $G = (V, E)$ where $V = \{-1, 0, 1, 2, \ldots, n * 2^n - 1\}$ and E consists of the edges

- $(i, i+1)$ for $i = -1, 0, 1, 2, \ldots, n * 2^n - 2$,
- (i, j) for $n2^n > j > i > 0$ and $p_i = p_j$,
- $(i, n * 2^n - 1)$ for i with $p_i = p_{n2^n}$,

and the charge function $f : V \to \{0, 1\}$ is defined by $f(-1) = 0$, $f(0) = 1$ and $f(i) = 0$ for $i > 0$. The observation that these sets of clauses coincide essentially goes back to [12].

The expansion $e(G)$ of an undirected graph $G = (V, E)$ is defined to be the smallest number

$$\#\{(v, v') \in E | (v \in B \wedge v' \notin B) \vee (v \notin B \wedge v' \in B)\}$$

for some $B \subseteq V$ satisfying $\frac{1}{3}\#V \leq \#B \leq \frac{2}{3}\#V$. For our graph $G = (V, E)$ the edges (v, v') satisfying $(v \in B \wedge v' \notin B) \vee (v \notin B \wedge v' \in B)$ correspond to pairs (i, j) as occurring in Lemma 9 up to a constant part of V. Hence by Lemma 9 we obtain $e(G) = \Omega(\#V/2n) = \Omega(2^n)$. In [1] the following two results were proved:

- Every resolution proof of $\tau(G, f)$ involves clauses with at least $e(G)$ literals.
- If a contradictory CNF on m variables of bounded clause size admits a resolution proof of length s, then it also admits a resolution proof only involving clauses of size $O(\sqrt{m \log s})$.

Hence, $\sqrt{n * 2^n * \log s} \geq c * 2^n$ for some $c > 0$, from which we conclude $s = 2^{\Omega(2^n/n)}$.

□

By using *expander graphs* it would be possible to prove the existence of contradictory biconditional formulas of size $\Theta(n)$ such that every resolution proof contains $2^{\Omega(i)}$ resolution steps. However, expressed in the size of the formula this improvement is only logarithmic compared to Theorem 10, while the construction of the formula is much more complicated.

6 The Main Result

We now have collected sufficient observations to come to our main result saying that the binary decision diagram technique is polynomially incomparable with any reasonable proof search technique based on resolution.

Theorem 11

- *There is a sequence of contradictory formulas ϕ_i of size $\Theta(i \log^2 i)$ $(i \geq 0)$ for which every OBDD proof has time and space complexity $O(i^2 \log^4 i)$, and for which each resolution proof requires $2^{\Omega(i)}$ resolution steps.*
- *There is a sequence of contradictory formulas ψ_i in CNF of size $\Theta(i^2)$ $(i \geq 0)$ that is proven in $O(i^2)$ steps using only unit resolution, and for which every OBDD proof has time and space complexity $\Omega(1.63^i)$.*

Proof:

- Take the formulas ϕ_i to be $\neg[S_n]$ from Theorem 10, where n is the smallest number satisfying $i \leq \frac{2^n}{n}$. Then the size of ϕ_i is $\Theta(n * 2^n) = \Theta(i \log^2 i)$, while by Theorem 10 every resolution proof requires $2^{\Omega(2^n/n)} = 2^{\Omega(i)}$ resolution steps. By Theorem 7 every OBDD proof has time and space complexity $O((n * 2^n)^2) = O(i^2 \log^4 i)^1$.
- Let ψ_i be $p \wedge (\neg p \wedge CR_{i,i})$. These formulas have size $\Theta(i^2)$. An OBDD proof of ψ_i contains an OBDD proof of $CR_{i,i}$ as one of its recursive calls; this takes time and space complexity $\Omega(1.63^i)$ by Theorem 5. It is easy to check that after applying the Tseitin transformation on ψ_i only unit resolution leads to a refutation in a number of steps linear in the size of ψ_i.

□

7 Further Research

In this paper we have shown that any technique based on a reasonable form of resolution is essentially different from the standard OBDD technique to prove formulas. However, many questions remain, such as:

1. Is there a natural strengthening of the resolution rule that allows us to simulate the construction of OBDDs polynomially by resolution? A good candidate is *extended resolution* (see e.g. [4]) where it is allowed to introduce new proposition letters defined in terms of existing ones. In [5] it has been shown that any system for propositional logic for which the soundness has a feasibly constructive proof, can be polynomially simulated by extended resolution. This holds for the OBDD method. A natural question is how to make such a simulation explicit.
2. On the other hand, there are modifications of the OBDD-technique by which for every formula ϕ the contrived example $p \wedge (\neg p \wedge \phi)$ can be handled efficiently, for instance the lazy strategy as described in [18]. How do these modifications of the OBDD-technique relate to resolution?
3. We have shown that biconditional formulas have short OBDD proofs, and after the Tseitin transformation they may require long resolution proofs. One can wonder whether contradictory conjunctive normal forms exist having polynomial OBDD proofs and requiring exponentially long resolution proofs. The Tseitin transformation of our biconditional formulas will not serve for this goal: OBDD proofs of these transformed biconditional formulas appear to be of exponential length.

References

1. Ben-Sasson, E., and Wigderson, A. Short proofs are narrow – resolution made simple. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing* (1999), pp. 517–526.
2. Bryant, R. E. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers C-35*, 8 (1986), 677–691.
3. Bryant, R. E. On the complexity of VLSI implmentations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers 40*, 2 (1991), 205–213.
4. Cook, S. The complexity of theorem proving procedures. *Proceedings of the 3rd annual ACM symposium on the Theory of Computing* (1971), 151–158.

[1] By a careful analysis using the specific structure of the formula $\neg[S_n]$ this can be improved to $O(i^2 \log^3 i)$.

5. Cook, S. Feasibly constructive proofs and the propositional calculus. *Proceedings of the 7th annual ACM symposium on the Theory of Computing* (1975), 83–97.
6. Davis, M., Logemann, G., and Loveland, D. A machine program for theorem proving. *Communications of the ACM 5* (1962), 394–397.
7. Goerdt, A. Davis-Putnam resolution versus unrestricted resolution. *Annals of Mathematics and Artificial Intelligence 6* (1992), 169–184.
8. Groote, J. F., van Vlijmen, S. F. M., and Koorn, J. W. C. The safety guaranteeing system at station Hoorn-Kersenboogerd. In *COMPASS-95, proceedings 10th annual Conference on Computer Assurance* (1995), IEEE, pp. 57–68.
9. Haken, A. The intractability of resolution. *Theoretical Computer Science 39* (1985), 297–308.
10. Marques Silva, J. P., and Sakallah, K. M. Grasp – a new search algorithm for satisfiability. Tech. Rep. CSE-TR-292-96, University of Michigan, Department of Electrical Engineering and Computer Science, 1996.
11. Meinel, C., and Theobald, T. *Algorithms and Data Structures in VLSI Design: OBDD — Foundations and Applications.* Springer, 1998.
12. Reckhow, R. *On the lengths of proofs in the propositional calculus.* PhD thesis, University of Toronto, 1975.
13. Stålmarck, G., and Säflund, M. Modeling and verifying systems and software in propositional logic. In *Safety of Computer Control Systems (SAFECOMP '90)* (1990), B. Daniels, Ed., vol. 656, Pergamon Press, pp. 31–36.
14. Tseitin, G. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logic, part 2* (1968), pp. 115–125. Reprinted in J. Siekmann and G. Wrightson (editors),*Automation of reasoning* vol. 2, pp. 466-483.,Springer–Verlag Berlin, 1983.
15. Uribe, T. E., and Stickel, M. E. Ordered binary decision diagrams and the Davis-Putnam procedure. In *First conference on Constraints in Computational Logic* (1994), J.-P. Jouannaud, Ed., vol. 845 of *Lecture Notes in Computer Science*, Springer, pp. 34–49.
16. Urquhart, A. Hard examples for resolution. *Journal of the ACM 34*, 1 (1987), 209–219.
17. Urquhart, A. The complexity of propositional proofs. *The Bulletin of Symbolic Logic 1*, 4 (1995), 425–467.
18. van de Pol, J. C., and Zantema, H. Binary decision diagrams by shared rewriting. In *Mathematical Foundations of Computer Science, MFCS2000* (2000), M. Nielsen and B. Rovan, Eds., vol. 1893 of *Lecture Notes in Computer Science*, Springer, pp. 609–618.

On Expressive Power of Second Order Propositional Program Logics*

Nikolai V. Shilov[1,2], Kwang Yi[1]

[1] Korean Advanced Institute of Science and Technology,
Taejon 305-701, Kusong-dong Yusong-gu 373-1,
Korea (Republic of Korea)
e-mail: {shilov, kwang}@ropas.kaist.ac.kr
[2] Institute of Informatics Systems,
Russian Academy of Sciences, Siberian Division,
6, Lavrentiev ave., 630090, Novosibirsk, Russia
e-mail: shilov@iis.nsk.su

Abstract. We examine expressive power of second order propositional program logics: logic $2M$ of C. Stirling and a new Second Order Elementary Propositional Dynamic Logic (SOEPDL). We demonstrate that SOEPDL is more expressive than $2M$, and them both are more expressive than the propositional μ-Calculus of D. Kozen (μC). We give also an "external" characteristic of SOEPDL expressive power: SOEPDL is as expressive as Second order Logic of monadic Successors of M. Rabin (**S(n)S**-Logic) in spite of different semantics. Thus, SOEPDL seems to be the most expressive state-based propositional program logic. Finally we discuss decidability issues of SOEPDL: undecidability of SOEPDL in general, but non-elementary decidability on infinite trees in particular (vs. elementary decidability of μC in all models and on infinite trees). We also give a new game-theoretic proof that μC is more expressive than a very popular with model checking community state-based Computation Tree Logic (CTL).

1 SOEPDL vs. Propositional Program Logics

The propositional μ-Calculus of D. Kozen (μC) [6, 7] is a powerful propositional program logic with fixpoints. In particular, a very popular with model checking community state-based temporal Computation Tree Logic (CTL) [4, 2, 3] is expressible in μC. We give a new proof that CTL is less expressive than μC:

Proposition 1.

1. *No CTL formula can express an existence of a winning strategy in finite games.*
2. *There is μC formula which expresses an existence of a winning strategy in finite games.*

But in spite of expressive power of μC , there exist more expressive propositional program logics. In particular, μC is expressible in the second order state-based program logic $2M$ of C. Stirling [14] while $2M$ is not expressible in μC:

Proposition 2.

1. *No μC formula can express in finite models Church-Rosser property for uninterpreted programs.*
2. *There is $2M$ formula which expresses Church-Rosser property for uninterpreted programs.*

We suggest a new Second Order Elementary Propositional Dynamic Logic (SOEPDL). The only difference between SOEPDL and $2M$ is interpretation of modalities $\Box/\Diamond$: in SOEPDL they mean "for every/some state" while in $2M$ they mean "for every/some reachable state". We demonstrate that $2M$ is expressible in SOEPDL while SOEPDL is not expressible in $2M$:

Proposition 3.

1. *No $2M$ formula can express in finite models the weakest preconditions for an uninterpreted postcondition and for backward computations of an uninterpreted program.*
2. *There is SOEPDL formula which expresses the weakest preconditions for an uninterpreted postcondition and for backward computations of an uninterpreted program.*

Thus, SOEPDL can be characterized as the most expressive state-based propositional program logic:

Theorem 1. *$CTL < \mu C < 2M <$ SOEPDL where all expressibilities have linear complexity and all inexpressibilities can be justified in finite models.*

* This work is supported by Creative Research Initiatives of the Korean Ministry of Science and Technology

2 SOEPDL vs. S(n)S-Logic

The "internal" characteristic of the expressive power of SOEPDL in terms of propositional program logics correlates with an "external" characteristic in terms of Second order logic of monadic Successors of M. Rabin (**S(n)S**-Logic) [8, 9, 1]. A comparison of SOEPDL and **S(n)S**-Logic is not straightforward, since SOEPDL and **S(n)S**-Logic have different semantics: SOEPDL is based on indivisible states while **S(n)S**-Logic is based on states generated by multiple first order variables, action symbols in SOEPDL are interpreted as binary relations in Kripke structures while functional symbols in **S(n)S**-Logic are interpreted as monadic successors in Herbrand model (i.e., full infinite **n**-fold trees).

The first problem can be resolved by consideration of formulae of **S(n)S**-Logic with a single (at most) free first order variable. In this case states generated by multiple first order variables can be identified with indivisible states presented by values of this single variable.

The last problem can be resolved by explicit references to classes of models: Herbrand models and Kripke structures. We would like to remark that Herbrand models are a particular case of Kripke structures.

Proposition 4. *For every SOEPDL formula it is possible to construct in linear time a formula of* **S(n)S**-*Logic with a single (at most) free first order variable such that both formulae are equivalent in Kripke structures.*

Proposition 5. *For every formula of* **S(n)S**-*Logic with a single (at most) free first order variable it is possible to construct in linear time SOEPDL formula such that both formulae are equivalent in Kripke structures.*

Combining propositions 4 and 5 we get

Theorem 2. *Expressive powers of SOEPDL and formulae of* **S(n)S**-*Logic with a single (at most) free first order variable are linear time equivalent in Kripke structures in general as well as in Herbrand models in particular.*

It is known that μC extended by a formula for commutativeness of computations is undecidable [11]. It is also known that **S(n)S**-Logic is non-elementary decidable in Herbrand models [8, 9, 1]. These facts together with theorems 1 and 2 imply the following

Corollary

1. *2M and SOEPDL are undecidable.*
2. *2M and SOEPDL are non-elementary decidable in Herbrand Models.*
3. *Lower bound for SOEPDL in Herbrand Models is non-elementary.*

We would like to remark also that in Herbrand models μC $\cong$ **S(n)S**-Logic [10] but this time an interpretation of **S(n)S**-Logic in μC has non-elementary complexity.

3 Conclusion

We have demonstrated that CTL $<$ μC $<$ 2M $<$ SOEPDL $\cong$ **S(n)S**-Logic and that SOEPDL is non-elementary decidable on infinite trees while is undecidable in general case. It is also well-known that μC is exponentially decidable on infinite trees as well as in general case [5]. In contrast to μC and SOEPDL, decidability bounds for 2M on infinite trees are still an open question. From one side, 2M is closely related to Quantified Propositional (linear) Temporal Logic (QPTL), and non-elementary decidability for QPTL has been established [13]. Simultaneously, 2M is closely related to another second order propositional program logic – Second Order Propositional Dynamic Logic (of program schemata) (SOPDL), and exponential upper bound for SOPDL on infinite trees has been proved in [11]. Thus more research are required for accurate decidability bounds of 2M on infinite trees.

Another possible dimension for comparisons of propositional program logics is model checking power. If LG is a logic and MD is a class of models, then a model checker for LG×MD is a program (algorithm) which can check LG formulae in MD models. Assume LG′ is a propositional program logic and MC′ be a model checker for LG′×MD. Assume we would like to check formulae of another program logic LG″ in models in MD. A first move is to try to reuse MC′, i.e., to force MC′ to do this job instead of expensive and risky design, implementation and validation of a new model checker MC″ for LG″×MD. If LG″ $\leq$ LG′ then the work is done. The question is: when LG″ $\not\leq$ LG′, is it still possible to reuse MC′ for LG″×MD? A forthcoming paper [12] demonstrates that CTL, μC , 2M, and SOEPDL have equal model checking power in every class of models MD, which contains the class of finite models and is closed with respect to Cartesian products and power-set operation.

References

1. Börger E., Grädel E., Gurevich Y. *The Classical Decision Problem.* Springer, 1997.
2. Burch J.R., Clarke E.M., McMillan K.L., Dill D.L., Hwang L.J. *Symbolic Model Checking: 10^{20} states and beyond.* Information and Computation, v.98, n.2, 1992, p.142-170.
3. Clarke E.M., Grumberg O., Peled D. Model Checking. MIT Press, 1999.
4. Emerson E.A. *Temporal and Modal Logic. Handbook of Theoretical Computer Science*, v.B, Elsevier and The MIT Press, 1990, 995-1072.
5. Emerson E.A., Jutla C.S. *The Complexity of Tree Automata and Logics of Programs.* SIAM J. Comput., v.29, n.1, 1999, p.132-158.
6. Kozen D. *Results on the Propositional Mu-Calculus.* Theoretical Computer Science, v.27, n.3, 1983, p.333-354.
7. Kozen D., Tiuryn J. *Logics of Programs.* Handbook of Theoretical Computer Science, v.B, Elsevier and The MIT Press, 1990, 789-840.
8. Rabin M.O. *Decidability of second order theories and automata on infinite trees.* Trans. Amer. Math. Soc., v.141, 1969, p.1-35.
9. Rabin M.O. *Decidable Theories.* in Handbook of Mathematical Logic, ed. Barwise J. and Keisler H.J., North-Holland Pub. Co., 1977, 595-630.
10. Schlingloff H. *On expressive power of Modal Logic on Trees.* LNCS, v.620, 1992, p.441-450.
11. Shilov N.V. *Program schemata vs. automata for decidability of program logics.* Theoretical Computer Science, v.175, n.1, 1997, p.15-27.
12. Shilov N.V., Yi K. *Model Checking Power of Propositional Program Logics.* Manuscript, 9p. (Available at URL `http://ropas.kaist.ac.kr/~shilov/fics.ps`)
13. Sistla A.P., Vardi M.Y., Wolper P. *The Complementation Problem for Büchi Automata with Applications to Temporal Logics.* Theoretical Computer Science, v. 49, 1987, p.217-237.
14. Stirling C. *Games and Modal Mu-Calculus.* Lecture Notes in Computer Science, v.1055, 1996, p.298-312.

An Extension of Dynamic Logic for Modelling OCL's @*pre* Operator

Thomas Baar, Bernhard Beckert, and Peter H. Schmitt

Universität Karlsruhe, D-76128 Karlsruhe, Germany
e-mail: {baar,beckert,pschmitt}@ira.uka.de

Abstract. We consider first-order Dynamic Logic with non-rigid functions, which can be used to model certain features of programming languages such as array variables and object attributes. We extend this logic by introducing an operator @*pre* on functions that makes a function after program execution refer to its value before program execution. We show that formulas with this operator can be transformed into equivalent formulas of the non-extended logic. We briefly describe the motivation for this extension, which is a related operator in the Object Constraint Language (OCL).

1 Introduction

Since the Unified Modeling Language (UML) has been adopted as a standard of the Object Management Group (OMG) in 1997, many efforts have been made to underpin the UML—and the Object Constraint Language (OCL), which is an integral part of the UML, —with a formal semantics. Most approaches are based on providing a translation of UML/OCL into a language with a well-understood semantics, e.g., BOTL [3] and the Larch Shared Language (LSL) [4].

Within the KeY project (see i12www.ira.uka.de/~key for details), we follow the same line, translating UML/OCL into Dynamic Logic (DL). This choice is motivated by the fact that DL can cope with both the dynamic concepts of UML/OCL and real world programming languages used to implement UML models (e.g. Java Card [2]).

The OCL allows to enrich a UML model with additional constraints, e.g., invariants for UML classes, pre-/post-conditions for operations, guards for transitions in state-transition diagrams, etc. Although, at first glance, OCL is similar to an ordinary first-order language, closer inspection reveals some unusual concepts. Among them is the @*pre* operator. In OCL, this unary operator is applicable to attributes, associations, and side-effect-free operations (these are called "properties" in the OCL context). The @*pre* operator may only be used in post-conditions of UML operations. A property *prop* followed by @*pre* in the post-condition of an operation $m()$ evaluates to the value of *prop* before the execution of $m()$.

Dynamic Logic [5–8] can be seen as an extension of Hoare logic. It is a first-order modal logic with modalities $[p]$ and $\langle p\rangle$ for every program p. These modalities refer to the worlds (called states in the DL framework) in which the program p terminates when started in the current world. The formula $[p]\phi$ expresses that ϕ holds in *all* final states of p, and $\langle p\rangle\phi$ expresses that ϕ holds in *at least one* of the final states of p. In versions of DL with a non-deterministic programming language there can be more than one such final state. There is no final state if p does not terminate. Deterministic programs have at most one final state. For these the formula $\phi \to [p]\psi$ is similar to the Hoare triple $\{\phi\}p\{\psi\}$.

We consider a version of first-order DL with non-rigid functions, i.e., functions whose interpretation can be updated by programs and, thus, can differ from state to state. Such non-rigid functions can be used to model features of real-world programming languages such as array variables and object attributes.

Moreover, to ease the translation of OCL into DL, we extend DL with an operator corresponding to OCL's @*pre*. The DL @*pre* operator makes a non-rigid function after program execution refer to its value before program execution. This allows to easily express the relation between the old and the new interpretation. For example, $[p](c \doteq c^{@pre})$ expresses that the program p does not change the interpretation of the constant c.

The main contribution of this paper is to show that a DL-formula with the @*pre* operator can be transformed into an equivalent formula without @*pre*.

The proofs for the stated theorems can be found in [1].

2 Dynamic Logic with Non-rigid Functions

Although non-rigid functions are mostly ignored in the literature, the more specific concept of array assignments has been investigated in [5, 7]. In both papers their semantics is handled by adding to each state valuations of second-order array variables. We introduce, instead, non-rigid function symbols. This shift of attention comes naturally when we want to axiomatise the semantics of object-oriented languages in DL. In this setting non-static attributes of a class are best modelled by non-rigid functions.

Let $\Sigma = \Sigma_{nr} \cup \Sigma_r$ be a signature, where Σ_{nr} contains the non-rigid function symbols and Σ_r contains the rigid function symbols and the predicate symbols, which are all rigid (Σ_r always contains the equality relation $\doteq$). The set $Term(\Sigma)$ of terms and the set $Fml_{FOL}(\Sigma)$ of first-order formulas are built as usual from Σ and an infinite set *Var* of object variables. A term is called *non-rigid* if (a) it is a variable or (b) its leading function symbol is in Σ_{nr}.

The set $Fml_{DL}(\Sigma)$ of DL-formulas is constructed as usual using the modalities $[p]$ and $\langle p \rangle$. In the following, we often do not differentiate between the modalities $\langle p \rangle$ and $[p]$, and we use $\{p\}$ to denote that it may be of either form.

We do not fix the syntax of the programs p. The set of allowed programs is denoted with $Prog_{DL}(\Sigma)$. A typical example is to take $Prog_{DL}(\Sigma)$ to be the set of *while* programs built with the programming constructs (generalised) assignment, *while* loop, *if-then-else*, and non-deterministic choice. Assignments are generalised in the sense that not only variables but also non-rigid terms $f(t_1, \ldots, t_k)$ may occur on the left-hand side (i.e., the value of non-rigid functions can be changed).

As usual, we use a Kripke-style semantics to evaluate the formulas in $Fml_{DL}(\Sigma)$ and the programs in $Prog_{DL}(\Sigma)$. The set of states of a *DL-Kripke structure* $\mathcal{K}$ is obtained as follows: Let $\mathcal{A}_0$ be a fixed first-order structure for the rigid signature Σ_r, and let A denote the universe of $\mathcal{A}_0$. An n-ary function symbol $f \in \Sigma_r$ is interpreted as a function $f^{\mathcal{A}_0} : A^n \to A$ and every n-ary relation symbol $r \in \Sigma_r$ is interpreted as a set $R^{\mathcal{A}_0} \subseteq A^n$ of n-tuples. A *variable assignment* is a function $u : Var \to A$. We use $u[x/b]$ (where $b \in A$ and $x \in Var$) to denote the variable assignment such that $u[x/b](y) = b$ if $x = y$ and $u[x/b](y) = u(y)$ otherwise; moreover, if V is a set of variables, then $u_{|V}$ denotes the *restriction* of u to V. The set S of all *states* of $\mathcal{K}$ consists of all pairs $(\mathcal{A}, u)$, where u is a variable assignment and $\mathcal{A}$ is a first-order structure for the signature Σ, whose reduction to Σ_r, denoted with $\mathcal{A}_{|\Sigma_r}$, coincides with $\mathcal{A}_0$.

The interpretation $\rho(p)$ of a program p is a relation on S. The semantics of DL-formulas, i.e., whether a formula ϕ is true in some state $(\mathcal{A}, u)$ (denoted by $(\mathcal{A}, u) \models \phi$), is defined as usual in modal logics using the accessibility relation $\rho(p)$ to interpret a modality $\{p\}$.

The results being proved in this paper hold true provided that the function ρ, which defines the semantics of the programs $p \in Prog_{DL}(\Sigma)$, satisfies the following two conditions. Let $\mathcal{K} = (S, \rho)$ be a DL-Kripke structure over signature Σ, let p be a program, and let V_p be the set of all variables occurring in p.

1. The program p only changes variables in V_p; that is, for all $((\mathcal{A}, u), (\mathcal{B}, w)) \in \rho(p)$, if $u(x) \neq w(x)$ then $x \in V_p$.
2. The domain of the relation $\rho(p)$ is closed under changing variables not in V_p in the following sense: If $((\mathcal{A}, u), (\mathcal{B}, w)) \in \rho(p)$ and $u'_{|V_p} = u_{|V_p}$, then there is a pair $((\mathcal{A}, u'), (\mathcal{B}, w')) \in \rho(p)$ with $w'_{|V_p} = w_{|V_p}$ and $u'_{|Var \setminus V_p} = w'_{|Var \setminus V_p}$.

These conditions are no real restrictions. They are, for example, met by the usual definition of ρ for *while* programs extended with the following semantics for generalised assignments: If p is of the form $f(t_1, \ldots, t_n) := s$, then $\rho(p)$ consists of all pairs $((\mathcal{A}, u), (\mathcal{B}, u))$ such that $\mathcal{B}$ coincides with $\mathcal{A}$ except for the interpretation of f, which is given by: $f^{\mathcal{B}}(b_1, \ldots, b_n) = s^{(\mathcal{A},u)}$ if $b_i = t_i^{(\mathcal{A},u)}$ for $1 \leq i \leq n$ and $f^{\mathcal{B}}(b_1, \ldots, b_n) = f^{\mathcal{A}}(b_1, \ldots, b_n)$ otherwise.

3 Dynamic Logic with the Operator @*pre*

We now define syntax and semantics of DL extended with the @*pre* operator, which can be attached to non-rigid function symbols. Intuitively, the semantics of $f^{@pre}$ within the scope of a modal operator $\{p\}$ is that of f before execution of p. If a formula contains nested modal operators, it may not be clear, to which state the @*pre* operator refers. To avoid confusion, we only allow @*pre* to be used in the Hoare fragment of DL, where formulas contain only one modal operator.

Definition 1. *The set $Term^{@}(\Sigma)$ of* extended terms *over $\Sigma = \Sigma_r \cup \Sigma_{nr}$ consists of all terms $t^{@}$ that can be constructed from some $t \in Term(\Sigma)$ by attaching @pre to arbitrarily many occurrences of function symbols from Σ_{nr} in t. Accordingly, the set $Form^{@}_{FOL}(\Sigma)$ of* extended first-order formulas *over Σ consists of all formulas $\phi^{@}$ that can be constructed from some $\phi \in Fml_{FOL}(\Sigma)$ by attaching @pre to arbitrarily many occurrences of function symbols from Σ_{nr} in ϕ.*

Definition 2. *The* Hoare fragment $H(\Sigma)$ *consists of all formulas of the form* $\forall z_1 \ldots \forall z_d(\phi \to [p]\psi)$ *where* $z_i \in Var$ $(0 \le i \le d)$, $p \in Prog_{DL}(\Sigma)$, *and* $\phi, \psi \in Fml_{FOL}(\Sigma)$.

The formulas in the extended Hoare fragment $H^{@}(\Sigma)$ *have the same form as those in the Hoare fragment* $H(\Sigma)$, *except that* $\psi \in Fml^{@}_{FOL}(\Sigma)$.

Note that a formula in the (extended) Hoare fragment does not have to be closed, i.e., $\{z_1, \ldots, z_d\}$ may be a *subset* of the occurring variables. In the following, we assume a formula of the form $\forall z_1 \ldots \forall z_d([p]\psi)$ to be an abbreviation of $\forall z_1 \ldots \forall z_d(true \to [p]\psi)$ and, thus, to be in the (extended) Hoare fragment.

Definition 3. *Let* $\mathcal{K}$ *be a DL-Kripke structure, let* $(\mathcal{A}, u)$ *be a state of* $\mathcal{K}$, *and let* $(\phi \to [p]\psi) \in H^{@}$. *The relation* $(\mathcal{A}, u) \models \phi \to [p]\psi$ *is defined in the same way as for formulas without @pre, except that, for any* $((\mathcal{A}, u), (\mathcal{B}, w))$ *in* $\rho(p)$, *the interpretation* $t^{(\mathcal{B},w)}$ *of the non-rigid terms in* $Term^{@}(\Sigma)$ *is given by:*

$$(f^{@pre}(t_1, \ldots, t_n))^{(\mathcal{B},w)} \quad = \quad f^{\mathcal{A}}(t_1^{(\mathcal{B},w)}, \ldots, t_n^{(\mathcal{B},w)}) \ .$$

In the following, we use notation like $(\mathcal{B}, w) \models \phi$ and $t^{(\mathcal{B},w)}$ for formulas ϕ resp. terms t containing the @*pre* operator if it is clear from the context which structure $\mathcal{A}$ is to be used for the interpretation of @*pre*.

4 Eliminating @*pre* Using Functions

After the pre-requisites we now define a translation function τ_f on the extended Hoare fragment that eliminates the @*pre* operator (the subscript f indicates that τ_f uses new function symbols). The idea of τ_f is to introduce, for each function f_i that occurs with the @*pre* operator, an associated new function symbol f^i_{pre} and to ensure that f^i_{pre} is interpreted in the right way. For example, the translation of $\langle p \rangle r(f_i^{@pre}(a))$ is $\forall x(f^i_{pre}(x) \doteq f_i(x)) \to \langle p \rangle r(f^i_{pre}(a))$.

This (rather naive) translation preserves universal validity of formulas (Theorem 1).

Definition 4. *Let* $\Sigma' = \Sigma'_r \cup \Sigma_{nr}$ *be an extension of signature* Σ *where* $\Sigma'_r = \Sigma_r \cup \Sigma_{pre}$ *and* Σ_{pre} *is disjoint from* Σ *and, for every* $f \in \Sigma_{nr}$, *contains a function symbol* f_{pre} *of the same arity as* f. *Then, the result of applying the* translation $\tau_f : H^{@}(\Sigma) \longrightarrow H(\Sigma')$ *to some formula* π *of the form* $\forall z_1 \ldots \forall z_d(\phi \to [p]\psi)$ *is*

$$\begin{array}{l} \forall z_1 \ldots \forall z_d \\ \quad ((\phi \wedge \bigwedge_{i=1}^{k} \forall x^i_1 \ldots \forall x^i_{n_i} f^i_{pre}(x^i_1, \ldots, x^i_{n_i}) \doteq \\ \qquad\qquad\qquad\qquad\qquad f_i(x^i_1, \ldots, x^i_{n_i}) \\ \quad) \to [p]\psi') \end{array}$$

where (a) $f_1, \ldots, f_k \in \Sigma_{nr}$ *are the function symbols occurring in* ψ *with attached @pre, (b)* $f^1_{pre}, \ldots, f^k_{pre}$ *are the corresponding function symbols in* Σ_{pre}, *(c) the* x^i_j *are pairwise distinct variables not occurring in the original formula* π, *(d)* ψ' *is the result of replacing all occurrences of* $f_i^{@pre}$ *in* ψ *by* f^i_{pre} $(1 \le i \le k)$.

Theorem 1. *Let* $\pi \in H^{@}(\Sigma)$. *Then,* $\models \pi$ *iff* $\models \tau_f(\pi)$.

Note, that the practical consequences of the above theorem are rather limited. Assume that Γ is a DL-formula without free variables and without @*pre*, and that π is a formula in the extended Hoare fragment for which we want to prove that $\Gamma \models \pi$. Because of the deduction theorem, that is equivalent to $\models \Gamma \to \pi$. Now, we would like to apply our translation τ_f to transform $\Gamma \to \pi$ into a formula without @*pre* and, making use of Theorem 1, prove the resulting non-extended formula instead. The translation τ_f, however, is only applicable if $\Gamma \to \pi$ is in the Hoare fragment, which requires Γ to be a pure first-order formula. This problem is avoided with our second translation presented in the following section.

5 Eliminating @*pre* Using Variables

Our second translation τ_v not only preserves validity but leads to a formula that is fully equivalent to the original one. However, it is only correct if the programs in $Prog_{DL}(\Sigma)$ are deterministic.

The basic idea of τ_v is to "flatten" all terms in a formula containing @*pre*. For example, $[p]r(f^{@pre}(a))$ is equivalent to $[p]\forall y(y \doteq f^{@pre}(a) \to r(y))$. This in turn is equivalent to

$$[p]\forall y_1 \forall y_2((y_1 \doteq f^{@pre}(y_2) \wedge y_2 \doteq a) \ \to \ r(y_1)).$$

Since y_1, y_2 are new variables and do not occur in p, the quantification can be moved to the front, and we get the formula $\forall y_1 \forall y_2 \{p\}((y_1 \doteq f^{@pre}(y_2) \wedge y_2 \doteq a) \rightarrow r(y_1))$. Finally, we have arrived at a point where we can eliminate the occurrence of @*pre* by moving the "defining" equality $y_1 \doteq f^{@pre}(y_2)$ of y_1 in front of the modal operator: $\forall y_1 \forall y_2 (y_1 \doteq f(y_2) \rightarrow (\{p\}(y_2 \doteq a \rightarrow r(y_1)))$. Note, that the "definition" $y_2 \doteq a$ of y_2 remains behind the modal operator because no @*pre* is attached to a.

Definition 5. *The* translation $\tau_v : H^{@}(\Sigma) \rightarrow H(\Sigma)$ *is defined for all formulas* $\pi = \forall z_1 \ldots \forall z_d (\phi \rightarrow \{p\}\psi)$ *from* $H^{@}(\Sigma)$ *as follows:*

Let $t_1, \ldots, t_l, \ldots, t_m, \ldots, t_k \in Term^{@}(\Sigma)$ *be all the (sub-)terms occurring in the formula* ψ $(1 \leq l \leq m \leq k)$*, where*

- *for* $1 \leq i \leq l$ *the term* $t_i = f_i^{@pre}(s_1^i, \ldots, s_{n_i}^i)$ *is not a variable and has the @pre operator attached to its leading function symbol,*
- *for* $l < i \leq m$ *the term* $t_i = f_i(s_1^i, \ldots, s_{n_i}^i)$ *is not a variable and does* not *have the @pre operator attached to its leading function symbol,*
- *for* $m < i \leq k$ *the term* t_i *is a variable.*

Then,

$$\begin{aligned} \tau_v(\pi) \; = \; & \forall z_1 \ldots \forall z_d \forall y_1 \ldots \forall y_k \\ & ((\phi \wedge \textstyle\bigwedge_{i=1}^{l} y_i \doteq f_i(x_1^i, \ldots, x_{n_i}^i)) \rightarrow \\ & \{p\}((\textstyle\bigwedge_{i=l+1}^{m} y_i \doteq f_i(x_1^i, \ldots, x_{n_i}^i) \wedge \\ & \textstyle\bigwedge_{i=m+1}^{k} y_i \doteq t_i) \rightarrow \psi')) \;, \end{aligned}$$ [1]

where (a) the $y_1, \ldots, y_k$ *are pairwise distinct variables not occurring in the original formula* π*, (b) for all* $1 \leq i \leq m$ *and* $1 \leq j \leq n_i$*, the variable* x_j^i *is identical to* y_{ind} *where* $ind \in \{1, \ldots, k\}$ *is the (unique) index such that* $t_{ind} = s_j^i$*, and (c)* ψ' *is the result of replacing, for* $1 \leq i \leq k$*, all occurrences of* t_i *in* ψ *on the top-level (i.e., not the sub-term occurrences) by* y_i*.*

Theorem 2. *Let all* $p \in Prog_{DL}(\Sigma)$ *be deterministic, i.e.,* $(s, s_1) \in \rho(p)$ *and* $(s, s_2) \in \rho(p)$ *implies* $s_1 = s_2$*.*
Then, $\models \pi \leftrightarrow \tau_v(\pi)$ *for all* $\pi \in H^{@}(\Sigma)$*.*

This theorem states the strongest result one could wish for. It implies that a Hoare fragment formula π can be substituted by $\tau_v(\pi)$ in any context. For instance, even if Γ is not a pure first-order formula, $\Gamma \rightarrow \pi$ can be translated into $\Gamma \rightarrow \tau_v(\pi)$. The reason why Theorem 2 requires the programs in $Prog_{DL}(\Sigma)$ to be deterministic is as follows: Intuitively, τ_v moves a universal quantification from behind the modal operator $\{p\}$ to the front of $\{p\}$. If the programs are non-deterministic, $\{p\}$ contains an implicit quantification over states. If $\{p\} = [p]$, that quantification is universal, and τ_v still works. If, however, $\{p\} = \langle p \rangle$, the translation τ_v intuitively moves a universal quantification over an implicit existential quantification, which is not correct. An example demonstrating that Theorem 2 does not hold for non-deterministic programs and the $\langle \cdot \rangle$ modality is given in [1]. Nevertheless, even if p is non-deterministic, τ_v can be used to remove the @*pre* operator from a formula π of the form $\phi \rightarrow \langle p \rangle \psi$ because π is equivalent to $\phi \rightarrow \neg [p] \neg \psi$ and, thus, to $\phi \rightarrow \neg \tau_v(true \rightarrow [p] \neg \psi)$. Then, however, the resulting formula is not in the Hoare fragment.

6 Summary

This paper demonstrates how the semantics of the OCL construct @*pre* can be integrated into an extended DL with non-rigid function symbols. Since the @*pre* operator is rather unusual, for practical reasons, it is useful to translate formulas with @*pre* into formulas without @*pre*. Our first translation τ_f only preserves validity of formulas, which in practice is often not sufficient. The second translation τ_v is more complex but leads to a fully equivalent formula. Both translations stay within the Hoare fragment, i.e., transform Hoare fragment formulas into Hoare fragment formulas. The translation τ_v can also be used to remove @*pre* from a non-Hoare formula π by applying it to all Hoare sub-formulas of π.

[1] If one of the variables y_i occurs in $\tau_v(\pi)$ on only one side of $\{p\}$, then $\tau_v(\pi)$ can be simplified by omitting the equality "defining" y_i and replacing all occurrences of y_i by the right side of that equality.

References

1. T. Baar, B. Beckert, and P. H. Schmitt. An extension of dynamic logic for modelling OCL's *@pre* operator. TR 7/2001, University of Karlsruhe, Department of Computer Science, 2001.
2. B. Beckert. A dynamic logic for the formal verification of Java Card programs. In *Proceedings, Java Card Workshop (JCW), Cannes, France*, LNCS. Springer, 2001. To appear.
3. D. Distefano, J.-P. Katoen, and A. Rensink. Towards model checking OCL. In *Proceedings, ECOOP Workshop on Defining a Precise Semantics for UML*, 2000.
4. A. Hamie, J. Howse, and S. Kent. Interpreting the Object Constraint Language. In *Proceedings, Asia Pacific Conference in Software Engineering*. IEEE Press, 1998.
5. D. Harel. Dynamic Logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic*. Reidel, 1984.
6. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
7. D. Kozen and J. Tiuryn. Logic of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 14, pages 89–133. Elsevier, 1990.
8. V. R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proceedings, 18th Annual IEEE Symposium on Foundation of Computer Science*, 1977.

Acknowledgement. We thank W. Ahrendt for fruitful discussions on the topic of this paper.

Event-Driven Re-Computation of Boolean Functions Using Decision Diagrams

Valeriy Vyatkin

Martin Luther University of Halle-Wittenberg,
Dept. of Engineering Science,
D-06099 Halle, Germany
phone: +49-(345)-552-5972, fax: +49-(345)-552-7304, e-mail: Valeriy.Vyatkin@iw.uni-halle.de

1 Motivation

This paper deals with the computational issues arisen in a class of real-time systems. A big deal of such systems as discrete controllers in industrial automation, have always been relying on massive Boolean computations. In these systems a controller communicates with the controlled system (called *plant*) by means of signals, which relay values of sensors to inputs of the controller and values of the outputs to the actuators of the plant. The controller is a computing device, implementing a control algorithm.

The control algorithm is usually represented in one of the programming languages specific for this field, implementation of which is reduced to the real-time computation of a (huge) number of Boolean functions.

The usual way of the computation is cyclic. First, the current status of the plant, which is indicated by sensors, is stored in an input buffer, then the whole control program (Boolean functions) is executed, while the values of inputs in the buffer remain unchanged, and in the last step the calculated outputs are transmitted to the actuators of the plant. Such a procedure, called *scan* repeats itself over and over again. The duration of the scan determines the response characteristic of controller. The shorter response is, the better the quality and reliability of the control are expected.

The latest trends in the development of control systems urgently require changes in these "cyclic" computations. We mention here two main reasons:

1. The control systems become distributed. As a consequence the data are delivered to/from controllers not directly, but via a network as events, i.e. messages about changes of the inputs/outputs. The new being developed standard for distributed controller design IEC61499, introduced in [4], largerly relies upon the event-driven implementation. This requires updated methods of computations.
2. The control algorithms themselves are getting more complicated. The supervisory control theory, introduced by Ramadge and Wonham [5], suggests to divide the controller onto two parts: the sequential controller which reflects the required processing cycle, and the supervisor, which observes the current situation and prevents the control system from getting to dangerous states. It is possible to build such supervisors automatically, given a formal model of a controlled plant and a formal description of the notion of "danger". The resulting supervisors, however, turn to be huge arrays of very complicated Boolean functions. Computation of supervisors is even more complicated, when the latter is placed to the distributed environment mentioned above.

In this paper we suggest a way of computation corresponding to the new challenges.

2 Re-Evaluation versus Evaluation

Binary Decision Diagram (BDD) is a directed acyclic graph (dag) with a single root, introduced in [6] for Boolean function representation. Evaluation of a function $f : \{0,1\}^n \to \{0,1\}$ is performed by a branching program with the structure of the BDD given an instance of input argument $X \in \{0,1\}^n$ in time $O(n)$ (for restricted BDD). This way of computation is also called start-over or full evaluation.

In this paper we introduce a BDD derivative termed Index Decision Diagram (IDD) which computes $f(X)$ in time linear to the "number of ones" in the vector X. Instead of dealing with the input vector represented as a Boolean vector, it is more compact to use its compressed form of the ordered list of indexes $\lambda(X)$ of the elements equal to 1. For example, $\lambda(\langle 0000100001\rangle) = \langle 5, 10\rangle$. The IDD is a BDD modification intended to process the input represented this way.

The IDD application can be beneficial for the computation if the input array is sparse, i.e. one value, for example zeros, predominate over the other (ones) in every input instance. The other application which we are going to show in this paper, is the event-driven reevaluation of a Boolean function. As opposed to the evaluation, the reevaluation finds $f(X_{new})$ given the change $\Delta \in \{0,1\}^n$ to the input X_{old} such that $X_{new} = X_{old} \oplus \Delta$. For example, $X_{old} = \langle 00010001 \rangle$, $\Delta = \langle 01000001 \rangle$, and $X_{new} = \langle 01010000 \rangle$. In many applications the change occurs just in a few bits of the input array at once, so Δ is a very sparse Boolean array and the IDD application seems to be reasonable.

When the reevaluation is applied to the event-driven systems it is assumed that the change Δ is induced by an event. We suggest to use some ***precomputation***, placed between events to prepare some auxiliary data which is used upon events to accelerate the ***on-line reevaluation***. It is important to minimize both parts, with the emphasis on the on-line part which is especially critical for the response characteristic of many discrete-event applications, such as logic control, etc. Usually in such applications events occur relatively seldom, so there is enough time for the pre-computations. However, once an event occurred the reaction must be as fast as possible. Certainly, reevaluation is worthwhile when compared to the evaluation if it can be performed in substantially less time than $O(n)$.

In this *event-driven* framework, the start-over algorithm can be regarded as having zero time precomputation and on-line reevaluation of $O(n)$ time. Another example of the event-driven algorithm, introduced in [7], precomputes $f(X_{old} + \Delta)$ for the subset $\{\Delta : |\lambda(\Delta)| \leq d\}$ and stores the precomputed values in the d-dimensional table. The precomputation takes $O(n^{d+1})$ time since the table has $O(n^d)$ entries and $O(n)$ time is required for each entry to be computed. Upon event, the value corresponding to the particular $\delta = \lambda(\Delta)$ can be restored from the table in time linear to the length of the list $O(|\delta|)(|\delta| \leq d)$.

The on-line algorithm presented in this paper uses the Index Decision Diagram to compute the result in time $O(|\delta|) = O(|\lambda(\Delta)|)$. Precomputation is used to compose the IDD given a BDD and values of arguments X. Upon event, the algorithm finds value $f(X_{old} \oplus \Delta)$ using the IDD and given δ. The problem of function's reevaluation is related also to the *incremental* methods of computations. More specifically, reevaluation using BDD is a particular case of the incremental circuit annotation problem [1,3].

3 Computation of Boolean Functions Using BDD

Let v_0 and V denote respectively the root and the set of vertices of a BDD. Each non-terminal vertex $v \in V$ has exactly two outgoing edges called 0– and 1– edge. When drawing a BDD, the 0-edge is depicted as a dotted line, and the 1-edge as a solid line. Target of the 0-edge is termed $lo(v)$ and of the 1-edge $hi(v)$. A non-terminal vertex is also associated with an input variable $x_{v.ind} \in X$ specified by the index $v.ind (1 \leq v.ind \leq n)$.

A BDD has exactly two terminal vertices, associated with constant $v.value \in \{0,1\}$ and assigned the pseudo index $v.ind \equiv n+1$. Each vertex v of the BDD denotes a Boolean function f_v as follows: in the terminal vertices $f_v \equiv v.value$, in the non-terminal ones it is defined in a recurrent way:

$$f_v = \overline{x_{v.ind}} f_{lo(v)} \vee x_{v.ind} f_{hi(v)}. \tag{1}$$

The function f_{v_0} denoted in the root is said to be denoted by the whole BDD. There is a two-way relationship between functions and diagrams — each Boolean function also can be expanded into a BDD by iterative application of the Shannon expansion [2].

A BDD is *restricted* (RBDD for short) if no variable is associated with more than one vertex in any directed path. Therefore length of any directed path in a RBDD is bounded by the size of the input array $|X| = n$. A BDD is *ordered* (OBDD) if in any directed path the indexes of the associated variables follow to the increasing order. Obviously, an ordered BDD is also restricted.

At a fixed input $X \in \{0,1\}^n$ one of the edges $(v, lo(v)), (v, hi(v))$ is called *active*. If $x_{v.ind} = 0$ then $(v, lo(v))$ is active, otherwise the opposite edge $(v.hi(v))$ is active. The destination of the active edges is called the active successor of the vertex: $w = active(v)$.

Let

$$active^i(v) = \begin{cases} active(v) & \text{if } \; i = 1 \\ active^{i-1}(v) & \text{if } \; i > 1 \end{cases}$$

denote the $i-th$ active successor of v. A directed path starting in v, formed only of active edges, and ending in a terminal vertex is denoted

$$v.\mathbf{P}_X = \langle v, active(v), active^2(v), \ldots, active^p(v) \rangle$$

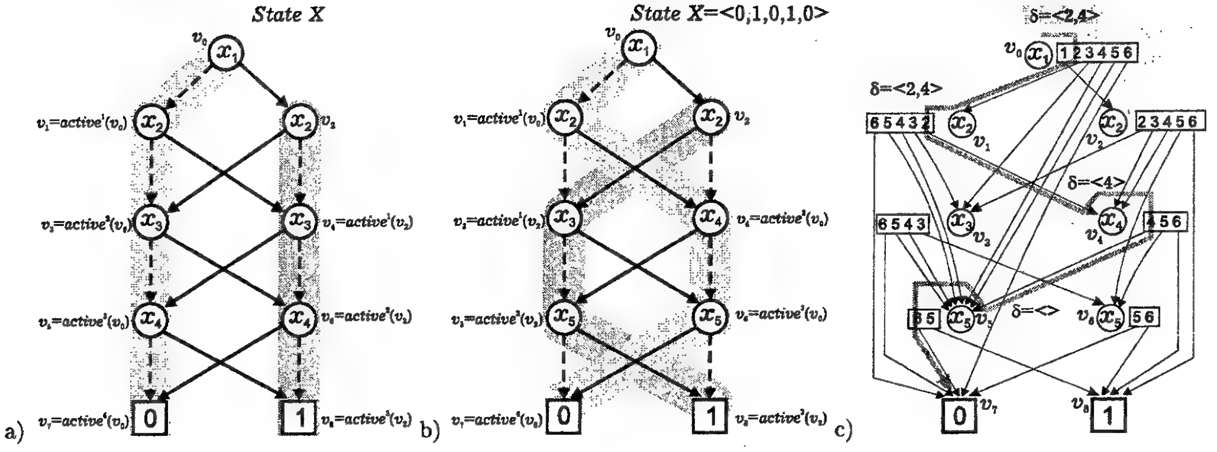


Fig. 1. Binary decision diagram with outlined active paths in the state $X = \langle 0,0,0,0\rangle$ (a) and after change $\delta = \langle 2,4\rangle$, i.e. at $X+\delta = \langle 0,1,0,1\rangle$ (b);Evaluation of IDD given the list $\lambda = \langle 2,4\rangle$ (c).

and called the active path of vertex v at input X. The subscript X emphasizes the dependence of the active path on the value of the current input. In particular if the input array contains only zeros, i.e. $X = \mathbf{0}$ the active path is called zero-path of the BDD. A vertex v is called *source of active path* if it has no active incoming edges. An active path which starts in a source vertex is called *full active path.*

Figure 1-a presents an example of OBDD for the function

$$f = \overline{(x_1 \oplus x_2)}(x_3 \oplus x_5) \vee (x_1 \oplus x_2)\overline{(x_4 \oplus x_5)}.$$

Active edges corresponding to $X = \mathbf{0}$ are gray shadowed. The full active paths in this state of X are rooted in v_0 and v_2: $v_0.\mathbf{P} = \langle v_0, v_1, v_3, v_5, v_7\rangle$, $v_2.\mathbf{P} = \langle v_2, v_4, v_6, v_8\rangle$ respectively.

The expression (1) can be transformed in terms of the active successor as follows:

$$f_v(X) = f_{active(v)(X)}. \tag{2}$$

The recursive computation of the function according to (2) can be performed by traverse of the BDD. The traverse starts in the root $v = v_0$ and always chooses the active child of the current vertex $active(v)$ to be continued. The value of the constant associated with the terminal vertex of the active full path $v.\mathbf{P}_X$ determines the current value of the function $f_v(X)$. Therefore, time of the full computation of function using RBDD is bounded by $O(n)$.

4 Index Decision Diagrams

Let $\lambda = \lambda(X)$ denote an ordered list of indexes of ones in X. For example if $X = \langle 0000100101\rangle$ then $\lambda(X) = \langle 5,8,10\rangle$. In this section we introduce the Index Decision Diagram (IDD) of a Boolean function which enables to compute $f(X)$ in $O(|\lambda(X)|)$ time given the list $\lambda(X)$.

Let G be an ordered BDD which denotes the function f, and $v.\mathbf{P_0}$ the *zero path.* We define the *search mapping* $M : V \times \{1,2,..,n\} \to V$ as follows: for given $v \in V$ and $\forall i \in v.ind \dots n+1$: $M_v(i)$ designates the vertex with minimum index greater than or equal to i in $v.\mathbf{P_0}$. The Index Decision Diagram (IDD) $\mathcal{E} = \mathcal{E}(G)$ is a graph which is built for a given BDD G as follows:

1. IDD and BDD have the same set of vertices $V_{\mathcal{E}} = V_G$.
2. A non-terminal vertex $v \in V_{\mathcal{E}}$ has $n - v.ind + 2$ outgoing edges defined by array of their targets $lihk_v[v.ind..n+1]$ such that: $link_v[v.ind] = hi(w)$ and the others $n - v.ind + 1$ edges are defined as $link[i] = M_v(i), i = v.ind+1,..,n+1$.

An example of IDD for the OBDD from Figure 1-a is presented in Figure 1-b. Note that since the zero-path with source v_0 does not contain vertex with variable x_4, the corresponding links in the vertices v_0, v_1, v_3 are redirected to the vertex associated with x_5. Similarly, in the zero-path with source in v_2 variable x_3 is not included and the link in v_2 is redirected to v_4.

```
Algorithm IDD evaluation ($\lambda$: list of integer)
begin
[1] $v$:=Root of the IDD;
[2] while $\lambda$ is not empty do
[3]         Loop invariant is: $\{v.ind \le min(\lambda)\}$
[4]         $v' := M_v(min(\lambda))$
[5]         Delete from $\lambda$ all numbers less than $v'.ind$
[6]         $v := v'$
[7] end{ while }
[8] if $v.ind < n+1$ then $v := M_v(n+1)$
[9] Result is $v.value$
end
```

Fig. 2. Algorithm reevaluates function using as input the IDD and the list λ of indexes of variables equal to 1.

The function denoted by the OBDD rooted in a vertex v depends on $x_{v.ind}, \dots, x_n$. Let us represent the input subarray $X[v.ind..n]$ as a concatenation of some "leading zero's" subarray $\mathbf{0}_{v.ind..i}$ and the remainder $X[i..n]$: $X[v.ind..n] = \mathbf{0}_{[v.ind..i-1]} \cdot X[i..n]$. If X is represented in such a way, i.e. if $x_{v.ind} = x_{v.ind+1} = .. = x_{i-1} = 0$, the following proposition holds:

Proposition 1. $f_v(X[v.ind..n]) = f_{M_v(i)}(X[i..n])$

Proof. Let us prove the statement by induction on i. If $i = v.ind$ then $M_v(i) = M_v(v.ind) = hi(v)$. According to (1) if $x_{v.ind} = 1$ then $f_v = f_{hi(v)}$ so the statement holds.

Assume that the statement holds for $i = k > v.ind$ and prove it for $i = k+1$. Denote $w = M_v(k)$. According to the definition of M_v, w is such that $w.ind$ is minimum $w.ind \ge k$.

If $w.ind = k$ then $lo(w).ind \ge k+1$ so $M_v(k+1) = lo(w)$. In case of $i = k+1$, $x_k = 0$ which implies $f_{M_v(k)} = f_w = f_{lo(w)} = f_{M_v(k+1)}(X[k+1..n])$.

If $w.ind > k$ (i.e $w.ind \ge k$) then $M_v(k+1) = M_v(k) = w$ and $f_{M_v(k+1)} = f_{M_v(k)}$.

Now suppose that the input vector is represented by the list $\lambda = \lambda(X)$. If root of the BDD (and IDD) is v then w.l.o.g. we can assume $X = X[v.ind..n]$ and $min(\lambda) \ge v.ind$. Then

$$X = \mathbf{0}_{[v.ind,min(\lambda)-1]} \cdot X[min(\lambda)..n]$$

and according to the proposition 1

$$f_v = f_{M_v(min(\lambda))}(X[min(\lambda)..n]).$$

The evaluation of $f(X)$ using IDD is performed by the algorithm presented in Figure 2. The input of the algorithm is list λ, the output is the value of the function. Main loop [2]-[6] iterates no more than $2|\lambda|$ times. In the body of the loop the current vertex v moves to $v' = M_v(\min(\lambda))$ in which $f_{v'} = f_v$. After that λ is adjusted to not contain any number less than $v'.ind$ so that the invariant $v.ind \ge min(\lambda)$ of the loop always holds. If the loop halts at $v : v.ind < n+1$ then there are no more indexes in λ which means that $x_{v.ind} = x_{v.ind+1} = .. = x_n = 0$. Therefore $f_v = f_v(\mathbf{0}_{v.ind..n}) = f_{M_v(n+1)}$. This is done in line [8].

Since the minimum of the ordered λ is $\lambda[1]$ and computation of $M_v(min(\lambda))$ is performed by the lookup in the linear array ***link*** in a constant time, the body of the loop takes constant time, so the total computation is done in time linear to $2|\lambda|$. A little modification of the mapping M_v is able to reduce the number of iterations to $|\lambda|$: assume $M'_v(i) = M_v(i)$ for all i such that the corresponding x_i is not presented in the zero-path $v.\mathbf{P}_0$, and $M'_v(i) = hi(M_v(i))$ if x_i is in the zero-path. Then each iteration of the loop [2-6] makes $min(\lambda) > v.ind$ which guarantees at least one index from λ to be deleted at each iteration thus bounding their number by $|\lambda|$. However we sacrifice this improvement to the clarity of explanation.

The algorithm of IDD evaluation is illustrated in Figure 1,c for $X = \langle 0, 1, 0, 1, 0 \rangle$, i.e. at $\lambda = \langle 2, 4 \rangle$. In the begin $v = v_0$,$min(\lambda) = 2$ and v moves to $v' = M_v(2) = v_1$. In $v = v_1$ we have $M_v(2) = v_4$ so that 2 to be deleted from λ since $v_4.ind = 4 > min(\lambda) = 2$. Then $v := M_{v_4}(4) = v_5$ and 4 is also to be deleted from λ. In this step the list λ is exhausted, so according to the line [9] of the algorithm the result can be found as *value* attribute of vertex $M_{v_5}(6) = v_8$.

The following theorem summarizes all stated above about the IDD evaluation:

Theorem 1. *The value of Boolean function $f(X)$ can be computed by the algorithm of IDD evaluation in time linear to the number of ones in the argument vector X using its presentation as an IDD.*

5 Generation of IDD

To build the IDD for a given OBDD G it is required to fill in every vertex $v \in V_G$ the arrays $link_v$ of pointers defining the edges going out of v in IDD. Complexity of this procedure is linear in the sum of links' number by all vertices of the IDD (and OBDD), and can be upper bounded by $O(n\,|V_G|)$.

6 Reevaluation of Boolean Function Using IDD

Let us denote $x^\alpha = \overline{x}$ if $\alpha = 1$ and $x^\alpha = x$ if $\alpha = 0$. Obviously, $0^\alpha = \alpha$ and $\alpha^\alpha = 0$. Let $X = \langle \alpha_1, \alpha_2, \ldots, \alpha_n \rangle$. The *normalized* function f_X is derived from a Boolean function f as follows:

$$f_X(x_1, x_2, \ldots, x_n) = f(x_1^{\alpha_1}, x_2^{\alpha_2}, \ldots, x_n^{\alpha_n}).$$

Value of the normalized function $f_X(\mathbf{0})$ with all-zeros argument $\mathbf{0} = 0, 0, \ldots, 0$ is equal to the $f(X)$:

$$f_X(\mathbf{0}) = f(0^{\alpha_1}, 0^{\alpha_2}, \ldots, 0^{\alpha_n}) = f(\alpha_1, \alpha_2, \ldots, \alpha_n).$$

The reevaluation is required to compute the function after a change Δ is occured with the input: $X_{new} = X \oplus \Delta$. The event-driven reevaluation consists of the on-line part which follows the event denoted as a list $\delta = \lambda(\Delta)$ and evaluates the new value of the function, and precomputation which is placed between events and provides the on-line part with required auxiliary data.

In our case the auxiliary data consists of the IDD for the normalized function f_X. It is clear that $f(X_{new}) = f_X(\Delta)$. As it follows from the previous section, having the IDD for the f_X the value of $f_X(\Delta)$ can be computed in time $O(|\lambda(\Delta)|)$. The OBDD for the normalized function can be derived from the OBDD denoting f trading places of its 0– and 1– edges in all the vertices $v : x_{v.ind} = 1$. It can be easily observed for a single-variable case with OBDD with one non-terminal vertex and then proved by induction for an arbitrary OBDD. Time of the transformation is bounded by $O(V)$. Given the OBDD for f_X, the IDD for f_X is composed in $O(n\,|V|)$ time, so the total precomputation required is bounded by $O(|V| + n\,|V|) = O(n\,|V|)$. Given the precomputed IDD and the list δ of indexes of changed variables the reevaluation requires $O(|\delta|)$ time to find the new value of the function.

We summarize this result as the following theorem:

Theorem 2. *On-line reevaluation of a Boolean function $f : \{0,1\}^n \to \{0,1\}$ after a change Δ to the input X can be done in time $O(|\lambda(\Delta)|)$ at the precomputation of $O(|V|\,n)$ time.*

References

1. B. Alpern, R. Hoover, B.K. Rosen, P.F Sweeney, and K. Zadeck. *Incremental evaluation of computational circuits*, pages 32–42. Proceedings of First Annual ACM-SIAM symposium on Discrete Algorithms, 1990.
2. C.E.Shannon. A symbolic analysis of relay and switching circuits. *Trans. AIEE*, 57:713–723, 1938.
3. Ramalingam G. *Bouded Incremental Computation*, volume 1089 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1996.
4. J.H.Christensen. Basic concepts of IEC 61499. In *Proc. of Conference "Verteile Automatisieriung" (Distributed Automation)*, pages 55–62, Magdeburg, Germany, 2000.
5. Ramadge P.J. and Wonham W.M. Supervisory control of a class of discrete event processes. *SIAM J Control Optimisation*, 25(1):206–230, 1987.
6. S.B.Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27:509–16, 1978.
7. V.Viatkin, K.Nakano, and T.Hayashi. *Optimized Processing of Complex Events in Discrete Control Systems using Binary Decision Diagrams*, pages 445–450. Int'l workshop on algorithms and architectures in real-time control, IFAC, 1997.

A Transformation of SDL Specifications — a Step towards the Verification

Natalia Ioustinova[1] and Natalia Sidorova[2]

[1] Dept. of Computer Science, Institut of Technische Informatik, University of Rostock,
Alb. Einstein Str. 21, D-18059 Rostock, Germany
fax +49(0)381 498 3440, ustin@informatik.uni-rostock.de

[2] Dept. of Math. and Computer Science, Eindhoven University of Technology,
PO Box 513, 5600 MB Eindhoven, The Netherlands
fax: +31(0)40 246 3992, e-mail: n.sidorova@tue.nl

Abstract. Industrial-size specifications/models (whose state space is often infinite) can not be model checked in a direct way — a verification model of a system is model checked instead. Program transformation is a way to build a finite-state verification model that can be submitted to a model checker. Abstraction is another technique that can be used for the same purpose. This paper presents a transformation of SDL timers allowing to reduce the infinite domain of timer values to a finite one with preserving the behaviour of the system. A timer abstraction is proposed to further reduce the state space. We discuss the ideas behind these transformations and argue their correctness.

1 Introduction

Model checking [3] is one of the most popular and successful verification techniques accepted both in academia and in industry. One of the main reason of its success is its promise to automatically check a program against a logical specification, typically a formula of some temporal logic. A stumbling-block limiting the model-checking application area is the notorious state-space explosion. The major factors influencing the state space are, clearly, the size and the complexity of a specification. In many cases, abstractions and compositional techniques make it possible to cope with the state-space explosion and apply model checking to real-life industrial systems. However, besides size and complexity, there exists another factor cumbering verification.

Development of a specification language, its semantical concept are greatly affected by the intended mode of its use, its application domain. Often, the final objective is to get an executable specification/implementation. In that case, the specification language and its semantics should provide a framework for constructing faithful descriptions of systems. No wonder that specifications written in these implementation-oriented languages are harder to verify than the ones written in the languages developed as input languages for model checkers. In this paper, we concentrate on some aspects of modelling time in the implementation-oriented languages, taking SDL (Specification and Description Language) [10] as an instance of this class of languages.

SDL is a popular language for the specification of telecommunication software as well as aircraft and train control, medical and packaging systems. Timing aspects are very important for these kinds of systems. Therefore, behaviour of a system specified in SDL is scheduled with the help of timers involved into the specification. The model of SDL timers was induced by manners of implementation of timers in real systems. An SDL timer can be activated by setting it to a value ($\texttt{NOW} + \delta$) where expression NOW provides an access to the current system time and δ is a delay after which this timer expires, i.e., the timer expires when a system time (system clock) reaches point ($\texttt{NOW} + \delta$). Such an implementation of timers immediately means that the state space of SDL-specifications is infinite just due to the fact that timer variables take an infinite number of growing, during the system run, values. An inverse timer model is normally employed in the verification-oriented languages: a timer indicates a delay left until its expiration, i.e., a timer is set to value δ instead of $(\mathtt{NOW} + \delta)$, and this value is decreased at every tick of the system clock. When the timer value reaches zero, the timer expires. This model of timers guarantees that every timer variable takes only a finite (and relatively small) number of values.

Another SDL peculiarity that adds to the complexity of verification is the manner the timers expire in SDL. SDL is based on the Communicating Extended State Machines; communication is organized via the message passing. For the uniformity of communication, timers are considered as a special kind of signals and a process learns about a timer expiration by dint of a signal with the name of the expired timer, inserted in the input port of the process. From the verification point of view it would be better if a timer expiration had been diagnosed by a simple check of the timer value.

Though formal verification of SDL-specifications is an area of rather active investigations [2, 7, 5, 9, 13], the time-concerned difficulties were being got round for a long time by means of abstracting out time and timers. Due to engineering rather than formal approaches to constructing abstractions, some of proposed abstractions turned out to be not safe (cf. [2]). In [2] a toolset and a methodology for the verification of time-dependent properties of SDL-specifications are described. The SDL-specifications are translated into DT Promela, the input language of the DT Spin (Discrete Time Spin) model checker [1], and then verified against LTL formulas. Some arguments are given in favour of a behavioural equivalence of the DT Promela translation to the original specification.

Here, we propose a transformation of SDL specification into SDL itself, where the new `timer` type is substituted for the traditional SDL `timer` type. The underlying idea is similar to the one in [2], but providing SDL to SDL transformation, we make the transformation principles independent of a particular model checker, and the formal proof of model equivalence substantiate that the transformed model can be safely used for the verification.

Admitting that in a number of cases timer abstractions are useful[1], we believe that the known safe abstraction (cf. [2]) of SDL timers does not yield a desirable result in a number of cases because it abstracts not just timers but time. Here, we propose a more flexible w.r.t. the refinement degree abstraction, for which the abstraction of [2] is a particular case.

The paper is organised as follows. In Section 2 we shortly survey the SDL time semantics. In Section 3 we present a behavioure-preserving transformation for SDL-specifications. In Section 4 we propose a timer abstraction. We conclude in Section 5 by evaluating the results.

2 SDL Time Semantics

SDL is a general purpose description language for communicating systems. The basis for the description of a system behaviour is Communicating Extended State Machines represented by processes. A process consists of a number of states and a number of transitions connecting the states. The input port of a process is an unbounded FIFO-queue. Communication is based on the signal exchange between the system and its environment and between processes within the system.

Two data types, `Time` and `Duration`, are used to specify time values. A variable of the `Time` type indicates some point of time. A variable of the `Duration` type represents a time interval. A process can access a current system time by means of the `NOW` operator. The concept of timers is employed to specify timing conditions imposed on a system. A timer is related to a process instance; it is either active (set to a value) or inactive (reset). Two operations are defined on the timers: `SET` and `RESET`. A timer is activated by setting it to a value $(\mathtt{NOW} + \delta)$ where δ is in fact a delay after which this timer expires.

With each timer there are associated a pseudo-signal and an implicit transition, called a time-out transition. When a timer expires, its time-out transition becomes enabled and may be executed. The execution of the time-out transition adds the corresponding pseudo-signal to the process queue. The time-out transitions of timers expiring simultaneously can be executed in any order. A time-out signal is handled like an ordinary signal. The timer is active until a pseudo-signal is consumed from the queue. If a `SET` or `RESET` operation is performed on an expired timer while its time-out transition is still enabled, the time-out transition becomes disabled. If the timer is set or reset after adding its associated pseudo-signal to the process queue (before the signal is consumed from the queue) the pseudo-signal is removed from the queue.

Though there is no standardized time semantics in SDL, there exist two semantics accepted in the SDL-community [5]. According to one of them (the one, which is supported by the commercial SDL-design tools [14, 12] and the one we work with), the transitions of SDL processes are instantaneous (take zero time); time can

[1] If a property is expected to hold independently of the settings of a timer, for example.

only progress if at least one timer is active and the system is blocked: all the processes are waiting for further input signals (i.e., all input queues are empty, except for saved signals, and there is no NONE input enabled). Time progression amounts to performing a specific transition that makes time increment until an active timer expires. Later on, we refer to a segment of time separated by the time increment transitions as a *time slice.* (Note that time progression is discretised.) When the system time gets equal to the timer value, the time-out transition becomes enabled and it can be executed at any point of the time slice. The time slice always starts with a firing of one of the enabled time-out transitions. This action unblocks the system. In case several time-out transitions become enabled at the same time, one of them is taken (non-deterministically) to unblock the system and the rest are taken later at any point of the time slice since they have the same priority as normal transitions.

Though complicated, such a time semantics is suitable for implementation purposes [10]. It is natural to model a timer as a unit advancing from the current time derived by evaluation of NOW expression to the the time point specified by the expression NOW + δ, i.e., waiting for a point of the system time.

3 Timer Transformation

An SDL process (its state) is described by its current control state, the states of the timers belonging to the process, the values of the process variables and the content of the input queue. Since NOW gives an access to the current system time, each evaluation of operation SET(NOW+δ,T) on timer T gives a new state of the process. Moreover, a time-out transition can add a timer signal at any point of the time slice. This blow up the state space due to the number of possible interleaving sequences of events. Keeping a time-out signal in a process queue also adds to the length of the state vector.

To avoid the state-space explosion due to the interpretation of timers and the overhead caused by the management of time-out pseudo-signals, we substitute SDL concept of timers as a special kind of signals by a concept of timers as a special data type. Timer variable T represents a declared in the original specification timer T. The value of this variable shows the delay left until the timer expiration. Since delays are non-negative, we use -1 to represent inactive timers. Therefore, the **RESET** operation is transformed into the assignment of the -1 value to a timer variable, and the SET(NOW+x,T) operation is transformed into the assignment of $\max\{0, x\}$ to it (Fig. 1).

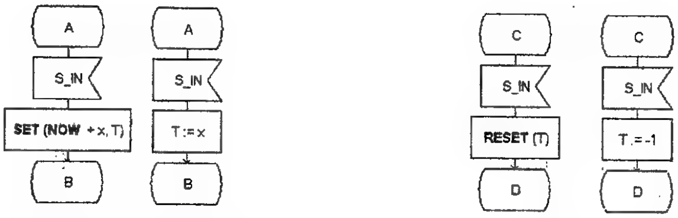


Fig. 1. Transforming **SET** (on the left) and **RESET** (on the right)

A timer whose value in the original system is equal to the current system time, can expire. The transformed system should demonstrate the same behaviour. Since we suppose that the value of the transformed timer is a delay left until its expiration, only the timers whose values are equal to 0 may expire. Therefore, we replace inputs of timer messages from the process queues by the enabling condition consisting of the NONE input guarded by the T=0 condition (Fig. 2).

Such a substitution does not give a straightforward reflection of the original system behaviour in the behaviour of the transformed system since the sending of the time-out signal to the process queue in the original system can be separated from its consumption from the queue by other actions. In the transformed system, we get the behaviour where sendings of time-out signals are projected out and the consumptions of these signals are mimicked by the consumptions of the correspondent NONE signals, whose enabling conditions are guaranteed to be true in this case. The projection is not harmful from the verification point of view because not the presence or absence of the time-out signal but the consumption of it and the resulted actions are important for the verification. The same concerns the process queue: saying that its content in the transformed system is the same as in the original one, we mean that the original queue from which the time-out signals are projected out, coincides with the process queue in the transformed system.

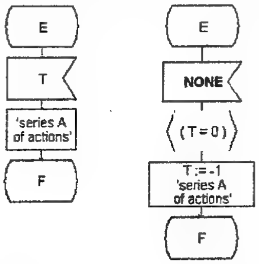


Fig. 2. Transforming the input of a time-out signal

Now, it is easy too see that the sequence of actions in the original system where a time-out signal is sent to the process queue due to a firing of the time-out transition and then removed from it because the correspondent timer is reset before the time-out signal is consumed from the queue, is fairly modelled in the transformed system. Neither sending nor removing the signal are not presented there being projected out, and the enabling condition may not be taken since the timer is reset, hence the condition is not fulfilled.

System time is not present in the transformed system — one infinitely grown variable is enough to cause the state-space explosion. The time-increment transition is mimicked by the transition decreasing the values of all active timers by the value of the minimal of them. Like the time-increment transition, this transition can take place only if the system is blocked, and "blocked" has exactly the same meaning as for the original system.

The equivalence of two models can be shown by using the simulation technique. Introducing the simulation relation that consider projected out timeout transitions and invisible ones, one can prove that the systems simulate each other in a step-wise manner by induction.

Corollary 1. *The transformation of SDL specifications according to the schema above preserves the system behaviour.*

4 Timer Abstraction

Abstraction, intuitively, means replacing one semantical model by an abstract, in general, simpler one. In addition to the requirement that an abstract (verification) model should have a smaller state space than the concrete (implementation) one, the abstraction needs to be *safe*, which means that every property checked to be true on the abstract model, holds for the concrete one as well[2]. This allows the transfer of positive verification results from the abstract model to the concrete one.

The concept of safe abstraction is well-developed within the *Abstract Interpretation* framework [4]. The requirement that Abstract Interpretation puts on the relation between the concrete model and its safe abstraction can be formalized as a requirement on the relation between the data operations of the concrete system and their abstract counterparts, as follows. Every value of the concrete state space is mapped by the *abstraction function* α into an abstract value which, intuitively, "describes" the concrete value. The requirement of mimicking is then formally phrased as:

$$\forall x : \alpha(f_{conc}(x)) \in f_{abs}(\alpha(x))$$

In the following we call this the *safety statement.*

Besides decreasing the state space of the system and safeness, the main requirement for an abstraction is that the abstract system behaviour should correctly reflect the behaviour of the original system with respect to a verification task in the sense that an abstraction captures all essential points in the system behaviour, i.e., it is not "too abstract". The safe abstraction of timers for the Promela translations of SDL-specification from [2] does not meet this requirement well enough.

The abstraction is based on a natural idea of allowing timers to expire at an arbitrary moment after they are set. A typical problem arising when one starts to apply this abstraction in practice is introducing "zero-time cycles" which are not present in the concrete model. A usual pattern for SDL-specifications is that a timer schedules some periodical activity; after a timer-out signal is consumed by the process, some actions are

[2] A safe abstract system is, intuitively, a system whose behaviour (the set of all transitions) is a superset of the concrete system behaviour.

initiated, then the timer is set again and the process returns to the same control state where it was before the consumption of the timer signal. Since a timer is allowed to expire at any arbitrary moment after its setting (also immediately after it has been set), an undesirable cyclic behaviour can be introduced. The "timer input–setting–expiration" chain of transitions is now permanently enabled and all the other behavioural branches may be ignored forever. Therefore, the properties which hold for the concrete model and which are expected to hold independently of the timer settings, fails to hold for the abstract model since "independently" means in fact that the property holds for a concrete model whatever *positive* delay is assigned to a timer. Another problem arises in case a timer serves as guard preventing from taking a transition too early. With abstracting time, this timer guard is broken.

We propose a safe abstraction for timers that keeps this guard delaying the timer expiration. A concrete timer that can be set only to delays exceeding some k is abstracted according to the patterns given in Figure 3. The setting of the concrete timer to a ($\mathtt{NOW}+x$) value is replaced with setting it to $\mathtt{NOW}+k$ ($x \leq k$) and the timer is allowed to expire at any point of time after the delay of k time units. This transformation only adds the behaviour while preserving all the behaviour of the concrete system, which means that the abstraction is safe. (The formal proof of safeness is performed via proving that the safety statement is not violated.) Varying k, we can change the refinement degree of the abstraction. Taking k equal to 0, we get the most abstract version of it, which is a particular case when not just timers but time is abstracted. Taking k equal to the lower boundary of the timer delays, we get the most refined abstraction defined by this transformation schema.

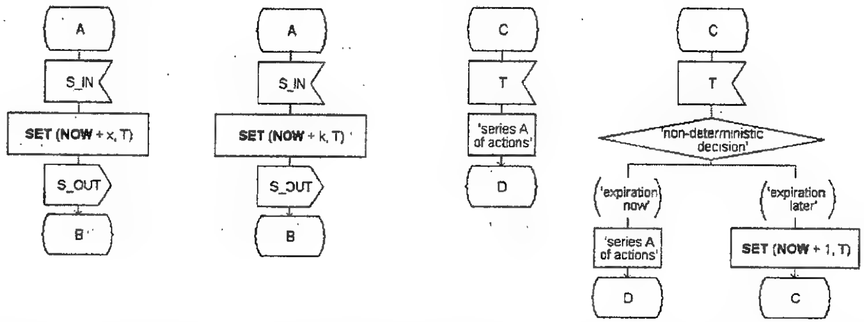


Fig. 3. Mimicking the `SET` operation (on the left) and the `INPUT` of a timer-signal (on the right)

The experiments shows that the smallest number of states is usually obtained when k is equal to 1, not 0. (We compare the number of states in the models, to which the transformation from Section 3 is applied.) The state space normally grows with increasing k, which is no wonder since the number of possible states of the timer itself grows in that case. Rather unexpected is the fact that 0 increases the state space and can even lead to the state-space explosion. The behaviour of the system with zero timer delay often has a different nature of regularity, the behaviour branches, excluded formerly by the timer guards, can be taken, which explains this phenomenon.

5 Conclusion

The proposed transformation of SDL-timers is a simple and cheap step that can be considered as a zero-phase of the translation of SDL-specifications to a model-checker input language. The transformation is aimed at reducing the state space to a finite domain. A side issue (though important on its own) is that the described transformation gives a better insight into the timer semantics. Our experience shows that the complicated time semantics of SDL can lead to errors due to its misunderstanding both in the design-phase and in the translation-phase (cf. [2, 11]). Treatment of timers as variables is simpler than treatment them as signals.

The abstraction given in Section 4 is aimed at the state space reduction. Due to its flexibility, it is applicable to a wider range of problems that the earlier used for SDL timer abstractions. Its practical application confirmed its efficiency for the real-life situations (cf. [11]).

Here, we gave only a sketchy description of the transformations of SDL-specifications and the intuitive ideas behind it. The formalization of the timer transformation and the timer abstraction will have to wait for the full paper, as well as the formal proof of the statements that the transformation does not change the specified system behaviour and that the abstraction is safe.

References

1. D. Bošnački, D. Dams, *Integrating Real Time into Spin: A Prototype Implementation*, S. Budkowski, A. Cavalli, E. Najm, editors, Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE/PSTV'98), Kluwer, 1998.
2. D. Bošnački, D. Dams, L. Holenderski, N. Sidorova, *Verifying SDL in Spin*, Tools and Algorithms for the Construction and Analysis of Systems TACAS 2000, LNCS 1785, pp. 363-377, Springer, 2000.
3. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.
4. P. Cousot, R. Cousot, *Abstract Interpretaion: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*, In the 4th POPL, Los Angeles, CA, ACM, January 1977.
5. U. Hinkel, *Verification of SDL Specifications on the Basis of Stream Semantics*, In Proc. of the 1st Workshop of the SDL Forum Society on SDL and MSC, Y. Lahav, A. Wolisz, J. Fischer, E. Holz (eds.), Humboldt-Universitaet zu Berlin.
6. G. J. Holzmann, *Design and Validation of Communication Protocols*, Prentice Hall, 1991.
7. G.J. Holzmann, J. Patti, *Validating SDL Specification: an Experiment*, In E. Brinksma, G. Scollo, Ch.A. Vissers, editors, Protocol Specification, Testing and Verification, Enchede, The Netherlands, 6-9 June 1989, pp. 317-326, Amsterdam, North-Holland, 1990.
8. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, S. Bensalem, *Property Preserving Abstractions for the Verification of Concurrent Systems*, In Formal Methods in System Design, Kluwer Academic Publ., 6, 1-36, 1995.
9. F. Regensburger, A. Barnard, *Formal Verification of SDL Systems at the Siemens Mobile Phone Department*, In Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98) 1998, LNCS 1384, pp. 439-455, Springer, 1998.
10. A. Olsen *et al.*, *System Engineering Using SDL-92*, Elsevier Science, North-Holland, 1997.
11. N. Sidorova, M. Steffen, *Verification of a Wireless ATM Medium-Access Protocol*, In Proc. of the 7th Asia-Pacific Software Engineering Conference (APSEC 2000), pp. 84-91, IEEE Computer Society, 2000.
12. Telelogic Malmö AB. *SDT 3.1 User Guide, SDT 3.1 Reference Manual*, Telelogic, 1997.
13. H. Tuominen, Embedding a Dialect of SDL in PROMELA, 6th Int. SPIN Workshop, LNCS 1680, pp. 245-260, Springer, 1999.
14. Verilog, *ObjectGEODE SDL Simulator - Reference Manual*, 1996.

Accurate Widenings and Boundedness Properties of Timed Systems

Supratik Mukhopadhyay and Andreas Podelski

Max-Planck-Institut für Informatik
Im Stadtwald, 66123 Saarbrücken, Germany
{supratik|podelski}@mpi-sb.mpg.de

Abstract. We propose a symbolic model checking procedure for timed systems that is based on operations on constraints. To accelerate the termination of the model checking procedure, we define history-dependent widening operators, again in terms of constraint-based operations. We show that these widenings are accurate, i.e., they don't lose precision even with respect to the test of boundedness properties.

1 Introduction

For the last ten years, the verification problem for timed systems has received a lot of attention (see e.g., [AD94,Bal96,DT98,LPY95,WT95]). The problem has been shown to be decidable in [AD94]. Most of the verification approaches to this problem have been based either on a region graph, which is a finite quotient of the infinite state graph, or on some variants of it (that use convex/non-convex polyhedra and avoid explicit construction of the full graph). But region-graph based approaches (or its variants) cannot be used for dealing with *boundedness* (unboundedness) properties (for definitions of these properties see an extended version of the paper available from http://www.mpi-sb.mpg.de/~supratik/mainwide.ps). This is due to the fact that the partitioning of the state space induced by the region equivalence (or any other technique that takes into account the maximal constant in the guards) is guaranteed to be *pre-stable* but may not be *post-stable.*

It can be shown that if the (symbolic) model checking algorithm in Figure 9 terminates, we can successfully model check for boundedness (unboundedness) properties. It is now natural to ask the question whether the procedure in Figure 9 is guaranteed to terminate. The answer is 'no'; consider the timed automaton in Figure 1 — the algorithm in Figure 9 will not terminate for this example (an infinite sequence of "states" which are not "included" in the "previously" generated states are produced). Of course, the procedure can be forced to terminate by including some maximal constant manipulation techniques (as the trim operation introduced in [MP00] or the extrapolation operation [DT98] or the preprocessing step [HKPV95]). But then, like the region graph technique, it can be shown that these techniques cannot be directly used for model checking for boundedness properties. So the natural thing now would be to develop techniques that force the termination of the procedure in Figure 9 (in cases where it is possible) but do not lose any information with respect to boundedness properties. It is in this context that *history-dependent constraint widenings* come into play.

Before introducing our framework of history-dependent constraint widenings (accurate widenings), let us try to see whether the already-existing *abstract interpretation* framework [CC77] can provide solutions to the problems described above. Abstraction interpretation techniques [CC77] are useful tools to force termination of the symbolic model checking procedures. Here one obtains a semi-test by introducing abstractions that yield a conservative approximation of the original property. Such methods have been successfully applied to many nontrivial examples [DT98,Bal96,WT95,HPR97]. While these abstractions force the termination of the model checking procedure, they sacrifice their accuracy in the process (note that by accuracy, we mean not only accuracy with respect to reachability properties, but also with respect to boundedness properties). One of the most commonly used abstractions is the *convex hull* abstraction [WT95,DT98,Bal96].

The application of automated, application independent abstractions that enforce termination, as is done in program analysis, to model checking seems difficult for the reason that the abstractions are often too rough[1]. To know the accuracy of an abstraction is important both conceptually and pragmatically. As Wong-Toi observes in [WT95],

> ...The approximation algorithm proposed is clearly a heuristic. It would be of tremendous value to have analytical arguments for when it would perform well, for when it would not....

[1] Note the statement of Halbwachs in [Hal93], that "Any widening operator is chosen under the assumption that the program behaves regularly.... Now the assumption of regularity is obviously abusive in one case: when a path in the loop becomes possible at step n, the effect of this path is obviously out of the scope of extrapolation before step n (since the actions performed on this path have never been taken into account)..."

As we saw above, any symbolic model checking procedure that "loses" accuracy will not be able to model check for boundedness (unboundedness) properties. Hence, in this paper, we propose a framework, to provide a partial answer to the question asked by Wong-Toi, viz., to determine automatically (using analytical methods) whether an abstraction performs well (does not lose accuracy) in a situation and then apply the abstraction.

We present methods that carry over the advantages of abstract interpretation techniques without losing precision. To be more specific, we apply history-dependent constraint widening techniques, as already foreseen in [CC77,CH78], to provide an application-independent abstract interpretation framework for model checking for timed systems. Basing our intuitions on techniques from Constraint Databases [JM94], we show that abstractions of the model checking fixpoint operator, through a set of widening rules, can yield an accurate model checking procedure. These abstractions are based on syntax of the constraints rather than their meaning (the solution space) in contrast with previous approaches (e.g., [Bal96,HPR97,WT95,BBR97]). As we demonstrate on examples, they can drastically reduce the number of iterations or even, in some cases, force termination of an otherwise non-terminating test. In contrast with the abstract interpretation techniques used for program analysis, they do not always force termination; instead their abstraction is accurate. That is, they do not lose information with respect to the original property; when they terminate, they provide information which is sufficient even for model checking for boundedness (unboundedness) properties; i.e., in cases where termination is achieved, the abstractions are sound and complete. Also, being based on the syntax of the constraints they can be implemented efficiently (they do not require computation of the convex hull like [WT95,Bal96,HPR97];). We first show toy examples in which our abstractions (henceforth called widening rules) either achieve termination in an otherwise non-terminating analysis or drastically accelerate the termination of symbolic forward reachability analysis.[2] We then show the performance of a prototype model checker, implemented using the techniques presented in this paper, on some standard benchmark examples taken from literature. In the Conclusion, we discuss the generality of our approach. The proofs and details are omitted from this extended abstract for lack of space. We invite the reader to go through an extended version of this abstract available at http://www.mpi-sb.mpg.de/~supratik/mainwide.ps.

2 Timed Automata, Constraints and Model Checking

For the purposes of this paper, we model timed systems using timed automata. We refer the reader to [AD94] for a formal treatment of timed automata.

We now fix the formal set up of this paper. We use lower case Greek letters for a constraint and upper case Greek letters for a set of constraints (which stands for their disjunction). The interpretation domain for our constraints is $\mathcal{R}$ the set of reals. We write $\mathbf{x}$ for the tuple of variables $x_1, \ldots, x_n$ and $\mathbf{v}$ for the tuple of values $v_1, \ldots, v_n$. As usual, $\mathcal{R}, \mathbf{v} \models \varphi$ is the validity of the formula φ under the valuation $\mathbf{v}$ of the variables $x_1, \ldots, x_n$. We formally define the relation denoted by a constraint φ as:

$$[\varphi] = \{\mathbf{v} \mid \mathcal{R}, \mathbf{v} \models \varphi\}$$

Note that $x_1, \ldots, x_n$ act as the free variables of φ and implicitly all other variables are existentially quantified. We write $\varphi[\mathbf{x}']$ for the constraint obtained by alpha-renaming from φ. We define $[\Phi]$, the relation denoted by a set of constraints Φ with respect to variables $x_1, \ldots, x_n$ in the canonical way. For a constraint φ and a set of constraints $\{\psi_1, \ldots, \psi_k\}$, we write $\varphi \models \bigvee_{i=1}^{k} \psi_i$ iff $[\varphi] \subseteq \bigcup_{i=1}^{k} [\psi_i]$. For sets of constraints Φ_1 and Φ_2 (where by a set of constraints $\Phi = \{\varphi_i\}$, we mean $\bigvee_i \varphi_i$), we write $\Phi_1 \models \Phi_2$ if for all $\varphi \in \Phi_1$ there exists $\varphi' \in \Phi_2$ such that $[\varphi] \subseteq [\varphi']$.We write an event (an edge transition or a time transition or a composition of several edge and time transitions) as **cond** ψ **action** φ, where the guard ψ is a constraint over $x_1, \ldots, x_n$ and the action φ is a constraint over the variables $x_1, \ldots, x_n$ and $x'_1, \ldots, x'_n$. The primed variable x' denotes the value of the variable x in the successor state. Note that we use interleaving semantics for our model. **We will use a set of constraints Φ to represent a set of states $\mathcal{S}$ if $\mathcal{S} = [\Phi]$.** The successor of a set of states of such a set with respect to an event $e \equiv$ **cond** ψ **action** φ is represented by the constraints obtained by conjoining the guard ψ and the action φ of each event with each constraint φ of Φ:

$$post_{|e}(\Phi) = \{\exists_{-\mathbf{x}'} \varphi \wedge \psi \wedge \varphi \mid \varphi \in \Phi, \mathcal{R} \models \varphi \wedge \psi \wedge \varphi\}$$

[2] Note that we consider forward analysis, instead of backward analysis, for the obvious advantages mentioned in [HKQ98] (Forward analysis is amenable to on-the-fly local model checking and also to partial order reductions. These methods ensure that only the reachable portion of the state space is explored). Moreover, backward analysis cannot be used for model checking for boundedness properties.

where the existential quantifier is over all variables but $\mathbf{x}'$.

We next formulate possibly non-terminating symbolic model checking procedures for boundedness properties, in our constraint-based framework, The template for the algorithm is given in Figure 9. Here $post(\Phi) = \cup_{e\in\mathcal{E}} post_{|e}(\Phi)$ where $\mathcal{E}$ is the set of all events of the timed system (*simple* and *compound*; see below for definitions of compound (composed) events). The algorithm is basically a (inflationary) fixpoint computation algorithm. Note that the template Symbolic-Boundedness can be used for model checking for the logic $\mathcal{L}_s$ [LPY95]. Also note that the algorithm is breadth first. In the sequel, we call the algorithm Symbolic-Boundedness as the breadth first (symbolic forward) reachability analysis algorithm.

The locations of a timed automaton can be encoded as finite domain constraints (in our algorithms we assume that the locations are encoded as finite domain constraints). We denote a position (simply a state) [AD94,HK97] of the timed automaton having location component ℓ as $\ell(\mathbf{v})$ where $\mathbf{v}$ denotes the values of the clocks. In general, for a set $\mathcal{S}$ of states having the location component ℓ, we write $\langle \ell, S\rangle$, or $\langle \ell(\mathbf{x}), \varphi\rangle$, where φ is a constraint and $S = [\varphi] = \{\mathbf{v} \mid \ell(\mathbf{v}) \in \mathcal{S}\}$. Here the free variables of φ are $\{x_1, \ldots, x_n\}$. In the sequel, we will refer to a set of states with location component ℓ and represented by $\langle \ell(\mathbf{x}), \varphi\rangle$ as a symbolic state or simply a state when it is clear from context.

3 Widening Rules

In this section, we consider how one can achieve (or just speed up) termination of the breadth first forward reachability analysis algorithms for boundedness (as well as safety) properties. We define widening rules that are accurate i.e., do not lose information with respect to the original property. We show that these widening rules can be used to achieve termination in cases where termination is not guaranteed in forward analysis. We also show that for some examples for which termination of forward analysis but widening can drastically accelerate the termination.

In general, the events considered here may not be an original event but is constructed as a composition of events. We write $e = event(\gamma, \varphi)$ when application of the event e to the constraint γ results in the constraint φ.

Given that the theory of reals with addition and order admits quantifier elimination, $\varphi \wedge \psi$ can be expressed in a conjunctive normal form.

We consider only non-strict inequalities here. The strict inequalities can be dealt with similarly. The template for symbolic boundedness procedure with widening is defined in Figure 4in the Appendix. The function $WIDEN$ is defined in Figure 5 in the Appendix. Note that the procedure in Figure 5 is based on a breadth-first search. In a call to $WIDEN(\Phi_i, post(\Phi_i))$ one of the three widening rules $WIDEN_1$, $WIDEN_2$ or $WIDEN_3$ described below is fired provided the conditions of that rule are satisfied. If the condition in the $WIDEN$ function applies to several decompositions of γ, the corresponding widenings are effectuated in several successive iterations. In the sequel, we refer to the procedure Symbolic-Boundedness-W as the breadth first forward reachability analysis procedure with widening.

We now illustrate the widening rules with examples. The intuition behind the widening rules is as follows: if we can detect from the syntax of a sequence of events $\bar{e}$ and a constraint φ, that the sequence $\varphi, post_{|\bar{e}}(\varphi), \ldots$ "grows" infinitely in a particular direction (i.e., actually leads to an infinite sequence with respect to reachability analysis), we will try to add the union of the sequence to our set of reachable states. Thus for widening rule I (for the if part), the syntax of the input constraint $(\eta \wedge x_i - x_j \geq c_{ij})$ and that of the event $(\theta \wedge x'_j = x_j + x'_i \wedge x_i \leq c_i$ which may be a composition of several simple events as described above) tells us that this constraint-event combination will generate an infinite behavior ($\eta \wedge x_i - x_j \geq c_{ij}$, $\eta \wedge x_i - x_j \geq c_{ij} - c_i, \ldots$; see example below) provided the other conditions are satisfied. Hence we infer the limit of this sequence which is η (since $c_{ij} \leq 0$ and $c_i > 0$) and add it to the set of states. Similar are the intuitions behind the other widening rules.

Consider the example timed automaton in Figure 1. Note that forward breadth-first reachability analysis does not terminate. Consider the events 4 and 3. Event 4 is given by $e \equiv$ **cond** $x_2 \leq 2$ **action** $x'_2 = 0, x'_1 = x_1$ (we do not show the location explicitly). Event 3 is the time event at location 1 and is given by $e' \equiv$ **cond** ***true*** **action** $x'_1 = x_1 + z, x'_2 = x_2 + z, z \geq 0$ (time increases by amount z). We compose transition (sometimes we will use the term 'transition' for 'event') 4 and transition 3 using the method given above. The resulting compound event is $e_1 \equiv$ **cond** ***true*** **action** $\varphi \wedge \psi$ where

$$\varphi \wedge \psi \equiv x'_1 = x_1 + x'_2, x_2 \leq 2, x'_2 \geq 0.$$

Now consider the infinite sequence of states produced by a breadth-first reachability analysis for this automaton

$$\begin{array}{l}
\langle l\_0(\mathbf{x}), x_1 = 0, x_2 = 0\rangle\mathsf{trans1}\langle l\_0(\mathbf{x}), x_1 = x_2, x_1 \geq 0\rangle \\
\mathsf{trans2}\langle l\_1(\mathbf{x}), x_1 = 0, x_2 \geq 0\rangle\mathsf{trans3}\langle l\_1(\mathbf{x}), x_2 - x_1 \geq 0, x_1 \geq 0\rangle \\
\mathsf{trans4}\langle l\_1(\mathbf{x}), 0 \leq x_1 \leq 2, x_2 = 0\rangle\mathsf{trans3}\ \overbrace{\langle l\_1(\mathbf{x}), x_1 - x_2 \geq 0, x_2 \geq 0, x_2 - x_1 \geq -2\rangle} \\
\mathsf{trans4}\langle l\_1(\mathbf{x}), 0 \leq x_1 \leq 4, x_2 = 0\rangle\mathsf{trans3}\langle l\_1(\mathbf{x}), x_1 - x_2 \geq 0, x_2 \geq 0, x_2 - x_1 \geq -4\rangle\mathsf{trans4}\ldots
\end{array}$$

(in the above we denote location i by l_i.) Now see that the state under the overbrace along with event e_1 satisfies the conditions of the widening rule I (the *if* part) defined in Figure 6 in the Appendix ($i = 2, j = 1$, $\gamma \equiv \eta \wedge x_2 - x_1 \geq -2$ where $\eta \equiv x_1 - x_2 \geq 0, x_2 \geq 0$, $c_{21} = -2$ and $\theta \equiv x'_2 \geq 0$). Hence, applying the widening, we obtain the state $\langle l_1(\mathbf{x}), x_1 - x_2 \geq 0, x_2 \geq 0\rangle$ (the reader can easily make out that if the sequence of transition 4 and transition 3 is repeated infinitely many times to the state under the overbrace, the constraint $x_1 - x_2 \geq 0, x_2 \geq 0$ will be obtained). After this any state generated is subsumed (included) by this state. Hence the breadth first forward reachability analysis with widening terminates.

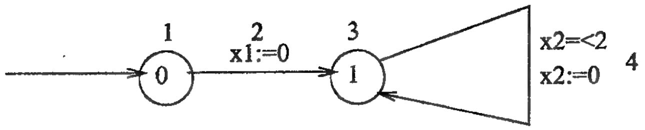


Fig. 1. Illustrating widening rule I

Before defining widening rule II, let us introduce some notation. Let $\mathcal{N}_n$ denote $\{1, \ldots, n\}$. Let I denote a subset of $\mathcal{N}_n$. The widening rule II is defined in figure 7 in the Appendix.

To show an example in which application of widening rule II forces termination, we look at the example in figure 2. Note that breadth-first forward reachability analysis does not terminate for this example. The following infinite sequence of states is generated in a breadth-first forward reachability analysis for this example.

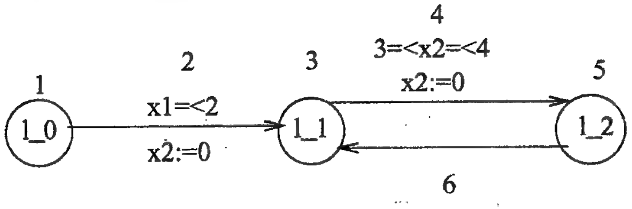


Fig. 2. Illustrating widening rule II

$$\begin{array}{l}
\langle l\_0(\mathbf{x}), x_1 = 0, x_2 = 0\rangle\mathsf{trans1}\langle l\_0(\mathbf{x}), x_1 = x_2, x_2 \geq 0\rangle \\
\mathsf{trans2}\langle l\_1(\mathbf{x}), x_1 \geq 0, x_1 \leq 2, x_2 = 0\rangle\mathsf{trans3}\langle l\_1(\mathbf{x}), x_1 - x_2 \geq 0, x_2 - x_1 \geq -2, x_2 \geq 0\rangle \\
\mathsf{trans4}\langle l\_2(\mathbf{x}), x_1 \geq 3, x_1 \leq 6, x_2 = 0\rangle\mathsf{trans5}\langle l\_2(\mathbf{x}), x_1 - x_2 \geq 3, x_2 - x_1 \geq -6, x_2 \geq 0\rangle \\
\mathsf{trans6}\ \overbrace{\langle l\_1(\mathbf{x}), x_1 - x_2 \geq 3, x_2 - x_1 \geq -6, x_2 \geq 0\rangle}\ \mathsf{trans3}\langle l\_1(\mathbf{x}), x_1 - x_2 \geq 3, x_2 - x_1 \geq -6, x_2 \geq 0\rangle \\
\mathsf{trans4}\langle l\_2(\mathbf{x}), x_1 \geq 6, x_1 \leq 10, x_2 = 0\rangle\mathsf{trans5}\langle l\_2(\mathbf{x}), x_1 - x_2 \geq 6, x_2 - x_1 \geq -10, x_2 \geq 0\rangle \\
\mathsf{trans6}\langle l\_1(\mathbf{x}), x_1 - x_2 \geq 6, x_2 - x_1 \geq -10, x_2 \geq 0\rangle\ldots
\end{array}$$

Now consider the compound event $e_2 \equiv$ **cond** *true* **action** $\varphi \wedge \psi$ obtained by composing transitions 3, 4, 5 and 6. Here

$$\varphi \wedge \psi \equiv x'_1 \geq x_1 - x_2 + x'_2 + 2 \wedge x'_1 - x'_2 \leq x_1 - x_2 + 3 \wedge x'_1 \geq x_1 + x'_2 \wedge x'_2 \geq 0.$$

See that the conditions of widening rule II (the *if* part) are satisfied for e_2 and the state under the overbrace in the sequence ($i = 2, j = 1, \eta \equiv x_2 \geq 0, c_{21} = -6 < 0, c_{12} = 3$ and $\theta \equiv x'_1 \geq x_1 - x_2 + x'_2 + 2, x'_1 \geq x_1 + x'_2, x'_2 \geq 0$). The reader can easily convince herself that the give state and event e_2 do not satisfy the conditions of widening rule I). Applying the widening, we obtain the state $\langle l\_1(\mathbf{x}), x_1 - x_2 \geq 3, x_2 \geq 0\rangle$ (viewing the constraint solving involved geometrically may provide better intuitions). The states which are further generated are subsumed by this state. So breadth-first forward reachability analysis with widening terminates after this. Note that in this case, application of abstract interpretation with the convex hull operator as is done in [WT95,Bal96,HPR97] would produce the state $\langle l\_1(\mathbf{x}), x_1 - x_2 \geq 0, x_2 \geq 0\rangle$. This can lead to 'don't know' answers to certain reachability questions (e.g., consider the reachability question whether the location l_1 can be reached with the values of the clocks satisfying the constraint $x_1 - x_2 > 2, x_2 - x_1 > -3, x_2 \geq 0$). As for the extrapolation abstraction [DT98], we have already stated in the Introduction that it is unsuitable for model checking for boundedness properties.

In widening rule III we use periodic sets following Boigelot and Wolper [BW94]. Due to space limitations we provide the definition of periodic sets in the Appendix.

The widening rule III is defined in Figure 8 in the Appendix, where the predicate $int(x)$ is true if and only if x is a nonnegative integer. Consider the example in Figure 3. Note that breadth-first forward reachability analysis does not terminate for this example. The following infinite sequence of states is generated in course of a forward (breadth-first) reachability analysis for this example:

$$\begin{array}{l}
\langle l\_0(\mathbf{x}), x_1 = 0, x_2 = 0\rangle \mathsf{trans1} \langle l\_0(\mathbf{x}), x_1 = x_2, x_2 \geq 0\rangle \\
\mathsf{trans2}\langle l\_1(\mathbf{x}), x_2 = 0, x_1 \geq 0, x_1 \leq 1\rangle \mathsf{trans3}\, \overbrace{\langle l\_1(\mathbf{x}), x_1 - x_2 \geq 0, x_2 - x_1 \geq -1, x2 \geq 0\rangle} \\
\mathsf{trans4}\langle l\_2(\mathbf{x}), x_1 \geq 4, x_1 \leq 5, x_2 = 0\rangle \mathsf{trans5}\langle l\_2(\mathbf{x}), x_1 - x_2 \geq 4, x_2 - x_1 \geq -5, x_2 \geq 0\rangle \\
\mathsf{trans6}\langle l\_1(\mathbf{x}), x_1 - x_2 \geq 4, x_2 - x_1 \geq -5, x_2 \geq 0\rangle \ldots
\end{array}$$

Now we compose transitions 4, 5 and 6. The compound event is $e_3 \equiv$ **cond** *true* **action** $\varphi \wedge \psi$ where

$$\varphi \wedge \psi \equiv x_1' = x_1 + x_2' \wedge x_2' \geq 0 \wedge x_2 = 4.$$

It is easy to see that the state under the overbrace in the infinite sequence along with event e_3 satisfies the conditions of widening rule III ($i = 2, j = 1, \eta \equiv x_2 \geq 0, c_{12} = 0, c_{21} = -1 < 0$). Hence, applying widening rule III we get the state $\langle l\_1(\mathbf{x}), \exists k \geq 0, int(k), x_1 - x_2 \geq k * 4, x2 - x_1 \geq -1 - k * 4\rangle$. The states further generated are subsumed by this state. So (breadth-first) forward reachability analysis terminates after applying the widening rule. Note that application of abstract interpretation with the convex hull operator [HPR97,Bal96,WT95] will produce the state $\langle l\_1(\mathbf{x}), x_1 - x_2 \geq 0, x_2 \geq 0\rangle$. Hence for certain reachability questions we can get a 'don't know' answer.

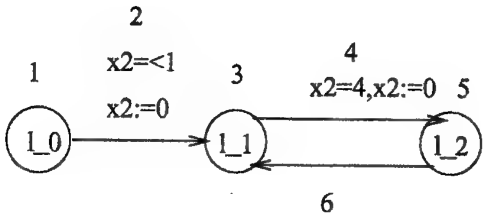


Fig. 3. Illustrating the widening rule III

Now we show that the widening rules are accurate with respect to boundedness properties.

Theorem 1 (Soundness and Completeness). *The procedure Symbolic-Boundedness-W obtained by abstracting the forward breadth first reachability analysis procedure with widening defined by the widening rules I,II and III yields (if terminating) a full test of boundedness (unboundedness) properties for timed systems (modeled by timed automata).*

▮

Note that the above theorem also implies that if the procedure Symbolic-Boundedness-W terminates, then one can get a full test of safety properties as well. Below we provide effective sufficient conditions for termination of Symbolic-Boundedness-W. By a *simple path* in a timed automaton $\mathcal{U}$, we mean a sequence of events $e_1 \ldots e_m$ where each e_i is an original event of $\mathcal{U}$ and

- the source location of e_{i+1} is the same as the target location of e_i for $1 \leq i \leq m - 1$,

- any event e_i with same source and target locations is a time event,
- for any two edge events e_i and e_j, $1 \leq i < j < m$, the target locations of e_i and e_j are different,
- and if e_i is an (original) time event, then e_{i-1} and e_{i+1} are edge events.

With this definition, there are only a finite number of such simple paths in a timed automaton. The simple path $p = e_1 \ldots e_m$ leads from location ℓ^1 to the location ℓ^2 if there is a the source location of e_1 is ℓ^1 and the target location of e_m is ℓ^2. The simple path $e_1 \ldots e_m$ is a simple cycle if the source location of e_1 is the same as the target location of e_m. Note that there are only a finite number of such simple cycles in a timed automaton.

Theorem 2 (Sufficient Conditions for Termination). *Let $\mathcal{U}$ be a timed automaton and let ℓ be a location in $\mathcal{U}$ such that there is a simple cycle C from ℓ to itself and the following three conditions are satisfied.*

- *There is a simple path in $\mathcal{U}$ of the form $e \equiv$ **cond** φ **action** ψ leading from the initial location ℓ^0 to ℓ such with the cycle C along with the the constraint $(\exists_{-\mathbf{x}'}\varphi^0 \wedge \varphi \wedge \psi)[\mathbf{x}]$ that satisfies the conditions of the widening rules I, II or III where φ^0 is the initial constraint.*
- *For each original event $e' \equiv$ **cond** φ' **action** ψ' with target location ℓ that lies on a cycle in the control graph of $\mathcal{U}$, $(\exists_{-\mathbf{x}'}\varphi' \wedge \psi')[\mathbf{x}] \models post_{|t}(\eta)$ if widening rule I or II is satisfied in the previous condition and $(\exists_{-\mathbf{x}'}\varphi' \wedge \psi')[\mathbf{x}] \models post_{|t}(\eta \wedge \exists k \geq 0 \wedge int(k) \wedge x_i - x_j \geq c_{ji} + k * c_j \wedge x_j - x_i \geq c_{ji} - k * c_j)$ if widening rule III is satisfied in the previous condition, where η, c_{ji} are as in the definition of the widening rules and t is the time event at location ℓ.*
- *The control graph of $\mathcal{U}$ satisfies the temporal formula $AG(true \Longrightarrow AF(at\_\ell))$ where at_ℓ is an atomic proposition satisfied only by location ℓ.*

Then the procedure Symbolic-Boundedness-W terminates for $\mathcal{U}$.

‖

It can be seen that the example in Figure 1 satisfies the sufficient conditions stated above.

We have implemented a prototype based on the approach (in the CLP($\mathcal{R}$) system of Sicstus Prolog 3.7). The performance shown, so far, by our approach has been quite encouraging. The experimental results are provided in an extended version of the paper available at http://www.mpi-sb.mpg.de/~mainwide.ps

References

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–236, 1994.

[Bal96] F. Balarin. Approximate reachability analysis of timed automata. In *17th IEEE Real-Time Systems Symposium*, pages 52–61. IEEE Computer Society Press, 1996.

[BBR97] B. Boigelot, L. Bronne, and S. Rassart. An improved reachability analysis method for strongly linear hybrid systems. In O. Grumberg, editor, *CAV'97: Computer Aided Verification*, volume 1254 of *LNCS*, pages 167–178. Springer-Verlag, 1997.

[BW94] Bernard Boigelot and Pierre Wolper. Symbolic verification with periodic sets. In David Dill, editor, *6th International Conference on Computer-Aided Verification*, volume 818 of *LNCS*, pages 55–67. Springer-Verlag, June 1994.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *the 4th ACM Symposium on Principles of Programming Languages*, 1977.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *the Fifth Annual ACM Symposium on Principles of Programming Languages.* ACM Press, 1978.

[DT98] C. Daws and S. Tripakis. Model checking of real-time reachability properties using abstractions. In Bernhard Steffen, editor, *TACAS98: Tools and Algorithms for the Construction of Systems*, LNCS 1384, pages 313–329. Springer-Verlag, March/April 1998.

[Hal93] N. Halbwachs. Delay analysis in synchronous programs. In C. Courcoubetis, editor, *the International Conference on Computer-Aided-Verification*, volume 697 of *LNCS*, pages 333–346. Springer-Verlag, 1993.

[HK97] Thomas. A. Henzinger and Orna Kupferman. From quantity to quality. In Oded Maler, editor, *Hybrid and Real-Time Systems International Workshop,Hart '97*, volume 1201 of *LNCS*, pages 48–62, Grenoble, France, March 1997. Springer-Verlag.

[HKPV95] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? In *the 27th Annual Symposium on Theory of Computing*, pages 373–382. ACM Press, 1995.

[HKQ98] T. A. Henzinger, O. Kupferman, and S. Qadeer. From pre-historic to post-modern symbolic model checking. In A. J. Hu and M. Y. Vardi, editors, *CAV'98: Computer-aided Verification*, LNCS 1427, pages 195–206. Springer-Verlag, 1998.

[HPR97] N. Halbwachs, Y-E. Proy, and P. Romanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.

[JM94] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19/20:503–582, May-July 1994.

[LPY95] K.G. Larsen, P. Pettersson, and W. Yi. Compositional and symbolic model checking of real-time systems. In *Proceedings of the 16th Annual Real-time Systems Symposium*, pages 76–87. IEEE Computer Society Press, 1995.

[MP00] S. Mukhopadhyay and A. Podelski. Model checking for timed logic processes. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *CL: Computational Logic*, LNCS, pages 598–612. Springer, 2000. Available at http://www.mpi-sb.mpg.de/~supratik/.

[WT95] H. Wong-Toi. *Symbolic Approximations for Verifying Real-Time Systems.* PhD thesis, Stanford University, 1995.

A Algorithms

```
**Procedure** Symbolic-Boundedness-W($\Phi$)
**Input** A set of constraints $\Phi$
**Output** A set of constraints representing sets of states reachable from $[\Phi]$
$\Phi_0 := \Phi$.
**repeat**
**begin**
$\Phi_{i+1} = \Phi_i \cup WIDEN(\Phi_i, post(\Phi_i))$
**end**
**until** $\Phi_{i+1} \models \Phi_i$.
return $\Phi_i$.
```

Fig. 4. Template for Model Checking for Boundedness Properties with Widening

```
**Function** $WIDEN(\Gamma, \Phi) = \{WIDEN(\gamma, \varphi) \mid \gamma \in \Gamma, \varphi \in \Phi\}$
**Function** $WIDEN(\gamma, \varphi)$
$\varphi_1 := WIDEN_1(\gamma, \varphi)$
**If** $\varphi_1 \not\equiv \varphi$ **return** $\varphi_1$
**else** $\{\varphi_1 := WIDEN_2(\gamma, \varphi)$
**If** $\varphi_1 \not\equiv \varphi$ **return** $\varphi_1$
**else** $\varphi_1 := WIDEN_3(\gamma, \varphi)\}$
**return** $\varphi_1$
```

Fig. 5. Widen Function

```
Function $WIDEN_1(\gamma, \varphi')$
if {
    $\gamma \equiv \eta \wedge x_i - x_j \geq c_{ij}$
    $\varphi \wedge \psi \equiv \theta \wedge x'_j = x_j + x'_i \wedge x_i \leq c_i$
    $c_{ij} \leq 0$
    $c_i > 0$
    $\eta[\mathbf{x}'] \models (\exists_{-\mathbf{x}'}(\eta \wedge \varphi \wedge \psi)) \wedge (\exists_{-\mathbf{x}'}\theta \wedge x_i - x_j \geq c_{ij} \wedge x_i \leq c_i)$
or if {
    $\gamma \equiv \eta \wedge x_i - x_j \geq c_{ij} \wedge x_i \leq c_i$
    $\varphi \wedge \psi \equiv \theta \wedge x'_j = x_j + x'_i$
    $c_{ij} \leq 0$
    $c_i > 0$
    $\eta[\mathbf{x}'] \models (\exists_{-\mathbf{x}'}(\eta \wedge \varphi \wedge \psi)) \wedge (\exists_{-\mathbf{x}'}\theta \wedge x_i - x_j \geq c_{ij} \wedge x_i \leq c_i)$
return $\eta$
else return $\varphi'$
```

Fig. 6. Widening Rule I

```
Function $WIDEN_2(\gamma, \varphi')$
if {
    $\gamma \equiv \eta \wedge x_i - x_j \geq c_{ij} \wedge x_j - x_i \geq c_{ji}$
    $\varphi \wedge \psi \equiv \theta \wedge x'_j - x'_i \leq x_j - x_i + a_{ji}$
    $c_{ij} \leq 0$
    $a_{ji} > 0$
    $(\exists_{-\mathbf{x}'}\varphi \wedge \psi \wedge \eta) \wedge (\exists_{-\mathbf{x}'}\theta \wedge x_i - x_j \geq c_{ij} \wedge x_j - x_i \geq c_{ji}) = \zeta \wedge x'_j - x'_i \geq c'_{ji}$
    $\eta[\mathbf{x}'] \models \zeta$
    $0 \leq c'_{ji} \leq -c_{ij}$
return $\eta \wedge x_j - x_i \geq c_{ji}$
else if {
    $\gamma \equiv \eta \wedge \bigwedge_{i,j,\in I} x_i - x_j \geq c_{ij}$
    $\varphi \wedge \psi \equiv \theta \wedge \bigwedge_{i,j,\in I} x_j - x_i \leq x_j - x_i + a_{ji}$
    $c_{ij} \leq 0$
    $a_{ji} > 0$
    $\eta[\mathbf{x}'] \models (\exists_{-\mathbf{x}'}\varphi \wedge \psi \wedge \eta) \wedge (\exists_{-\mathbf{x}'}\theta \wedge \bigwedge_{i,j,\in I} x_i - x_j \geq c_{ij})$
return $\eta$.
else return $\varphi'$
```

Fig. 7. Widening Rule II

```
Function $WIDEN_3(\gamma, \varphi')$
if {
    $\gamma \equiv \eta \wedge x_i - x_j \geq c_{ij} \wedge x_j - x_i \geq c_{ji}$
    $\varphi \wedge \psi \equiv \theta \wedge x'_i = x_i + x'_j \wedge x_j = c_j$
    $c_{ij} \leq 0$
    $c_i > 0$
    $c_{ji} \geq c_{ij}$
    $\eta[\mathbf{x}'] \models (\exists_{-\mathbf{x}'}(\eta \wedge \varphi \wedge \psi)) \wedge (\exists_{-\mathbf{x}'}\theta \wedge x_i - x_j \geq c_{ij} \wedge x_j - x_i \geq c_{ji} \wedge x_j = c_j)$
return $\eta \wedge \exists k \geq 0 \wedge int(k) \wedge x_i - x_j \geq c_{ji} + k * c_j \wedge x_j - x_i \geq c_{ji} - k * c_j$
else return $\varphi'$
```

Fig. 8. Widening Rule III

```
**Procedure** Symbolic-Boundedness($\Phi$)
**Input** A set of constraints $\Phi$
**Output** A set of constraints representing sets of states reachable from $[\Phi]$
$\Phi_0 := \Phi$.
**repeat**
**begin**
$\Phi_{i+1} = \Phi_i \cup post(\Phi_i)$
**end**
**until** $\Phi_{i+1} \models \Phi_i$.
return $\Phi_i$.
```

Fig. 9. Template for Model Checking for Boundedness Properties

Adaptive Saturation-Based Reasoning

Alexandre Riazanov and Andrei Voronkov

University of Manchester
e-mail: {riazanov,voronkov}@cs.man.ac.uk

Abstract. We describe the Limited Resource Strategy intended to improve performance of resolution-based theorem provers when a fixed limit is imposed on the time of a run. We give experimental evidence that the Limited Resource Strategy gives a significant improvement over the algorithms not using passive clauses for simplification and the weight-based algorithms.

1 Reasoning with Limited Resources

In nearly all applications, provers for first-order logic are used in the following way. A time limit is set for every particular goal, and if neither proof nor countermodel could be found within the time limit, the prover is terminated. Then the goal can be reconsidered, for example by formulating intermediate statements (lemmas) or by providing some inference steps interactively, and the proof-search continues on the new goals or using the lemmas. Setting a time limit for processing a particular goal is a natural idea, for most applications it is difficult to expect human users or systems ready to wait for an answer forever. It turns out that when the time is limited, systems can perform much better by using algorithms other than ordinary complete ones. In this abstract we describe such an algorithm, the so-called Limited Resource Strategy (LRS), implemented in our system Vampire [6], discuss its advantages and drawbacks and compare it with the strategies so far used in Vampire and other systems.

2 Saturation-Based Theorem Proving

The fastest first-order theorem provers of the last two CASC competitions [9] (with one exception of Setheo [5]) use saturation algorithms. There exist two main kinds of saturation algorithms, one was implemented in OTTER [4] and its predecessors (see [3]), another one was used for the first time in DISCOUNT [1]. Apart from OTTER, the former algorithm is implemented at least in Gandalf [10], SPASS [11], and Vampire, and the DISCOUNT algorithm has been adopted by E [7], SPASS and Vampire. Saturation algorithms used in first-order theorem provers operate on *clauses.* For each new clause generated by an inference the prover decides whether this clause should be *kept* or discarded. The set of kept clauses may be huge, so most of the systems perform inferences not on *all* kept clauses, but only on a subset of them. The clauses in this subset, i.e. those used for inferences will be called *active*, and all other kept clauses *passive.* The two different saturation algorithms differ in their treatment of passive clauses. In the DISCOUNT algorithm passive clauses never participate in inferences or simplifications, while in the OTTER algorithm passive clauses can participate in simplifications, such as rewriting by unit equalities or subsumption.

2.1 The OTTER Saturation Algorithm

The OTTER algorithm shown in Figure 1 is parametrized by several procedures explained below:

- *select* is the clause selection function. It decides which clause should be selected for activation.
- *infer* is the function that returns the set of clauses obtained by all inferences between the current clause *current* and the set of active clauses *active*. Usually, *infer* applies inferences in some complete inference system of resolution with paramodulation.
- *simplify*(*set*, *by*) is a procedure that deletes redundant clauses from *set* (e. g. those subsumed by clauses in *by* and tautologies) and simplifies some clauses in *set* using the clauses in *by* (e. g. rewritten by unit equalities in *by*). The simplified clauses are always moved to *passive*.
- Likewise, *inner_simplify* simplifies clauses in *new* using other clauses in *new*.

```
input: $init$ : set of clauses ;
var $active, passive, new$ : sets of clauses ;
var $current$ : clause ;
$active := \emptyset$ ;
$passive := init$ ;
while $passive \neq \emptyset$ do
  $current := select(passive)$ ;
  $passive := passive - \{current\}$ ;
  $active := active \cup \{current\}$ ;
  $new := infer(current, active)$ ;
  if $goal\_found(new)$ then return $provable$ ;
  $inner\_simplify(new)$ ;
  $simplify(new, active \cup passive)$ ;
  if $goal\_found(new)$ then return $provable$ ;
  $simplify(active, new)$ ;
  $simplify(passive, new)$ ;
  if $goal\_found(active \cup passive)$ then return $provable$ ;
  $passive := passive \cup new$
od ;
return $unprovable$
```

Fig. 1. The OTTER Saturation Algorithm

When we simplify *new* using the clauses in *active* ∪ *passive*, we speak of *forward simplification*, when we simplify *active* and *passive* using the clauses in *new*, we speak of *backward simplification*.

Typical behavior of this algorithm is quantitatively characterized by the following empirical observation: in a matter of seconds the total number of kept clauses gets very big, whereas the share of the active clauses is small and keeps decreasing. To illustrate this, we provide statistics on an unsuccessful run of Vampire with the time limit of 1 minute on the TPTP problem ANA003-1. During this run, 261,573 clauses were generated. The overall number of active clauses was 1,967, the overall number of passive clauses 236,389. The clauses generated in this run contain function symbols and equality, many of the clauses have a large number of literals and/or heavy terms. Even when the state-of-the-art term indexing techniques are used, it is very difficult to manage clause sets containing over 100,000 clauses efficiently. As a consequence, when theorem provers are used for practical applications, completeness is often compromised in favor of efficiency: the provers discard clauses that may be nonredundant.

2.2 The DISCOUNT Saturation Algorithm

It was observed that usually the total number of active clauses is considerably less than the number of passive clauses. Therefore, processing the passive clauses consumes a significant amount of time. One can modify the OTTER saturation algorithm in such a way that passive clauses never participate in simplifications. Such a modified saturation algorithm will be called *the DISCOUNT algorithm* in this abstract. The algorithm is shown in Figure 2.

Compared to the OTTER saturation algorithm, this algorithm has the following features:

- The new clauses are forward simplified by the active clauses only, the passive clauses do not take part in this.
- Neither active nor passive clauses are backward simplified by the retained new clauses.
- After selection of the current clause it is simplified again by the active clauses and then is itself used to simplify the active clauses only.

If we assume that the overall number of kept clauses is significantly larger than the number of used ones, this algorithm involves less computation for the same number of active clauses than the OTTER algorithm. However, this algorithm has a severe weakness: it performs a very limited amount of backward simplification steps compared to the OTTER algorithm. This sometimes results in the following effect: finding some proofs that are quickly found by the OTTER algorithm involving simplification, is now delayed significantly. For example, suppose that two unit clauses $t = a$ and $t = b$ are generated, where t is a heavy term, a and b are constants. If the prover using the DISCOUNT algorithm tries to select lighter clauses first, it may take a long time before

```
input: $init$ : set of clauses ;
var $active$, $passive$, $new$ : sets of clauses ;
var $current$ : clause ;
$active := \emptyset$ ;
$passive := init$ ;
while $passive \neq \emptyset$ do
  $current := select(passive)$ ;
  $passive := passive - \{current\}$ ;
  $simplify(\{current\}, active)$ ;
  if $current$ is simplified to a goal return $provable$
  if $current$ is not simplified to a tautology
  then do
    $simplify(active, \{current\})$ ;
    if $goal\_found(active)$ then return $provable$ ;
    $active := active \cup \{current\}$ ;
    $new := infer(current, active)$ ;
    if $goal\_found(new)$ then return $provable$ ;
    $simplify(new, active)$ ;
    $passive := passive \cup new$
  od ;
od ;
return $unprovable$
```

Fig. 2. The DISCOUNT Saturation Algorithm

these clauses become active. In the OTTER algorithm, rewriting $t = b$ by $t = a$ immediately gives a very light clause $a = b$ which is very likely to contribute to a derivation of the empty clause.

It was observed experimentally that the time spent for storing and retrieving passive clauses in the DISCOUNT algorithm is negligible compared to the overall runtime. Therefore, one cannot expect to improve considerably the performance of the DISCOUNT algorithm by, e.g. trying to discard some passive clauses when a time limit is set (though it can save a lot of memory).

3 Reasoning in Limited Time by the OTTER Algorithm

Growth of the number of kept clauses in the OTTER algorithm causes fast deterioration of the rate of processing of active clauses. Thus, when a complete procedure based on the OTTER algorithm is used, even passive clauses with high selection priority often have to wait indefinitely long before they contribute to the search. In the provers based on the OTTER algorithm, all solutions to the completeness-versus-efficiency problem are based on the same idea: some nonredundant clauses are discarded from the clause sets *active*, *passive*, or *new*. In this section we explain several approaches to discarding clauses implemented in the state-of-the-art provers and analyze their main advantages and disadvantages.

3.1 Weight Limit Strategy

The Weight Limit Strategy was implemented already in the very first versions of OTTER. The idea is to set a limit W on the weight of clauses. The weight of a clause is a measure reflecting its complexity, for example the number of symbols in it. All new clauses with the weight greater than W are discarded. This *Weight Limit Strategy* may be helpful for interactive use and solving difficult problems when the user can analyse the output of an unsuccessful run to adjust the weight limit, but it is not very useful for completely automatic theorem proving since there is no general method for choosing appropriate weight limit. Too small a limit leads to loss of proofs, too high a limit does not improve performance significantly compared to the complete algorthm. Another problem with the weight limit was observed by the authors in case studies: for many problems heavy clauses are needed for a very short time in the beginning of the proof-search, and then only very light clauses suffice for finding a proof.

3.2 Incremental Weight Limit Strategy

Several provers, including Gandalf [10], Bliksem [2] and Fiesta adopted the *Incremental Weight Limit Strategy.* In this strategy the weight limit is initially set to a small value. If no proof is found with this small value, the weight limit is increased, and the proof search begins either from scratch or using the short clauses obtained during the previous run.

3.3 Memory Limit Strategy

This strategy was implemented for the first time in OTTER. The idea is as follows. Some memory limit is set in advance. When $\frac{1}{3}$ of the available memory has been filled, OTTER assigns new weight limit which is calculated in such a way that 5% of passive clauses have smaller weight than the limit. From then on, this recalculation of weight limit is performed after processing every 10 selected clauses.

The main problem with this strategy is that the use of memory and the time are loosely connected. Setting too low a memory limit makes the prover terminate before the time limit because all clauses needed for finding a proof have been discarded. Setting too high a limit results in considerable slowdown, since then the system behaves as poorly as based on a complete algorithm.

4 Limited Resource Strategy

The main idea of LRS is the following. The system tries to identify which clauses in *passive* and *new* are *unreachable*, i.e. have no chance to be processed by the time limit at all, and discards these clauses. The notion of unreachable clauses is fundamentally different from the notion of redundant ones: redundant clauses are those that can be discarded without compromising completeness at all, the notion of unreachable clauses makes sense only in the context of reasoning with limited resources. How can one identify unreachable clauses? When the system starts solving a problem, it is given a time limit t as an argument. The system keeps track of statistics on the average time spent by processing each clause selected as *current*. Usually this time increases because the sets *active* and *passive* are growing, so the operations with them take more and more time. From time to time the system tries to estimate how the proof search statistics would develop towards the time limit and, based on this estimation, identify unreachable clauses.

The main requirements we imposed on the implementation are the following.

Requirement 1. Vampire with LRS should be at least as fast as Vampire using the complete algorithm.

Requirement 2. A proof should not be lost when the time limit is set to an acceptable value. Suppose that Vampire with a time limit t_1 has found a proof in time $t_2 < t_1$. Then Vampire with the time limit t_2 should find a proof as well. In other terms, when the time limit is set to t_2 no reachable clause should be lost.

Requirement 3. As many unreachable clauses as possible should be identified as unreachable by Vampire.

Requirement 3 is in conflict with Requirement 2, because the exact estimation of which clauses are unreachable is essentially impossible.

The following example gives the reader an idea how an estimation of unreachable clauses can be done. Suppose that p clauses have been processed as *current* in t seconds, i.e. p/t clauses per second, and l is the current time limit. If we assume that the proof search will develop at the same pace, in total $p \cdot l/t$ clauses will be processed by the end of the time limit. So if the number of currently kept clauses k is greater than $p \cdot l/t$, then $k - p \cdot l/t$ clauses can be discarded. Of course this estimation may be inaccurate, because the time for processing one clause as *current* will most likely increase. To avoid too big errors, the estimation of p/t must be done frequently enough. In our experiments the estimation was performed after every 500 inferences produced.

The next question is which $k - p \cdot l/t$ clauses should be discarded. The answer can be obtained by applying Requirement 2. One of the consequences of this principle is that no clause processed by the time limit by the complete strategy should be discarded. But the clause selection in the complete algorithm is controlled by the function *select*, so to identify potentially unreachable clauses let us look deeper at the clause selection function.

All modern theorem provers maintain one or more priority queues from wich clauses are picked using some *ratios.* By far the most popular design is based on two priority queues: the *age priority queue* gives higher priority to older clauses, the *weight priority queue* to lighter clauses. The rational behind this strategy is based on the following observation: light clauses are easy to process and most likely to contribute to a derivation of the empty clause, but discarding an old clause is more likely to turn an unsatisfiable set of clauses into a satisfiable one than for a younger clause. The system uses a ratio to decide how often the first clause in each queue should be selected. This ratio is called the ***pick-given*** ratio in OTTER's manual [4], we will call it the *age-weight ratio.* For example, if the age-weight ratio is 1:4, then out of each 5 selected clauses 1 will be taken from the age priority queue and 4 from the weight priority queue. This strategy was introduced for the first time in OTTER and then used by a number of systems.

Assume that our clause selection function is based on the age-weight queue design with age-weight ratio $a : w$, which means that out of any $a + w$ clauses a will be selected from the age queue and w from the weight queue. We have decided that $p \cdot (l/t - 1) = p \cdot (l - t)/t$ currently passive clauses can still be processed within the time limit. Of these clauses $a \cdot p \cdot (l - t)/(t \cdot (a + w))$ will be selected from the age priority queue and $w \cdot p \cdot (l - t)/(t \cdot (a + w))$ from the weight priority queue. So Vampire implements a deletion algorithm that discards clauses according to these formulas.

This example shows that LRS can delete many unreachable clauses from ***passive***, but it does not demonstrate the full power of the strategy. Suppose that the strategy discarded some clauses, and the maximal weight of the remaining clauses is W. Suppose a new clause C obtained by an inference has a weight $W' \geq W$. This clause is unreachable since it cannot be inserted in the reachable part of the weight priority queue and is younger than any clause in ***passive***. This means that any future clause with the weight $\geq W$ can be discarded, so we can set the limit on the weight to be $W - 1$.

Apart from using the dynamically changing weight limit W for discarding new clauses, we can also use the weight limit to discard any ***kept*** clause if any inference with this clause as a parent gives a clause with a weight exceeding W. Resolution-based provers use calculi based on ordered resolution with negative selection. A typical inference rule in such a calculus is ordered resolution with negative selection:

$$\frac{A \vee C \quad \neg B \vee D}{(C \vee D)\theta},$$

where θ is a most general unifier of A and B, the atom $A\theta$ is maximal in the clause $(A \vee C)\theta$ and $\neg B$ is a literal selected in the clause $\neg B \vee D$. The nonmaximal part of $A \vee C$ will always be part of the clause $C \vee D$. The application of the substitution θ to $C \vee D$ yields a clause at least as heavy as $C \vee D$ (unless we factor equal literals). Suppose now that we perform a resolution inference with the clause $A \vee C$. If the weight of C is greater than the weight limit, then any clause inferred from $A \vee C$ would be too heavy. Therefore, $A \vee C$ can be discarded from the search space because it cannot produce a reachable clause. To implement this, when LRS reduces the weight limit, we can search through the whole set ***passive*** $\cup$ ***active*** for clauses whose nonmaximal (nonselected) part has a weight greater than W and discard them.

When the weight of C does not exceed W we can sometimes simply compute the weight of $C\theta$. For example, if we have found the substitution θ together with a sufficiently big set of clauses containing the literal $\neg A\theta$, it still might be useful to compute the weight of $C\theta$ and compare it with W in an attempt to avoid building all the inferences. Moreover, to estimate weight of $C\theta$ it is often sufficient to have θ constructed only partially. This can be done, for example, when retrieval of literals unifiable with $\neg A$ is being performed in an index, in which case we can identify a branch in the index that does not have to be inspected.

5 Comparison of LRS with Other Approaches

The main feature of the LRS over other algorithms is the possibility to adapt to a particular problem based on the runtime information about the proof-search process. No previous knowledge about the problem is needed. In this section we briefly explain some advantages of LRS as compared to other existing approaches.

Weight-limit based approaches. Setting a particular weight limit in the beginning of proof-search can hardly be helpful since the weight limit needed to solve an unknown problem can not be calculated a priori. So we only compare LRS with the Incremental Weight Limit Strategy. This strategy has several well-known pitfalls. Suppose that the strategy is applied to a problem for which the minimal weight limit sufficient to solve the problem is W.

- For some problems, the proof-search with weight limits smaller than W can consume more time than the time limit, while setting the limit to W would solve the problem almost immediately. For such problems the Incremental Weight Limit Strategy is likely to be much less efficient than the complete strategy.
- For some problems, clauses with the weight W are only needed very early in the proof-search, and then clauses with weights less than or equal to some $W' < W$ will suffice. If too many clauses with the weights between W' and W are generated, the strategy can spend too much time on processing these clauses and will fail to find a proof.

The Limited Resource strategy is immune to both kinds of problems. For the first kind of problems, LRS will behave like a complete algorithm, since early in the proof-search LRS behaves like a complete strategy. For the second kind of problems, when LRS discovers that clauses of the weights between W' and W are unreachable, it will discard these clauses.

The DISCOUNT algorithm. The DISCOUNT algorithm behaves poorly for problems which require many backward simplification steps. Backward simplification steps are performed only when the simplifying clause becomes active. As a consequence, sometimes the algorithm cannot find proofs easily found by other strategies, especially when the proofs contain simplification steps between heavy clauses.

Shortcomings of LRS. Ideally, the requirements for LRS guarantee that it should not lose proofs found by a complete strategy. In reality, mistakes in calculating unreachable clauses are unavoidable, so in practice on some problems the complete algorithm beats LRS.

The main reason for miscalculating reachability of clauses is backward simplifications. When an LRS-based algorithm discards clauses beyond the dynamically set weight limit, it is possible that a simplification of a discarded clause would result in a short proof.

However, our experiments carried out over a large number of problems demonstrate that on the average the performance of the LRS-based algorithm is superior to other algorithms.

6 Experiments

To compare the LRS-based algorithm with the DISCOUNT algorithm, we implemented the DISCOUNT algorithm in Vampire and made a number of experiments on two benchmarks suites: (i) all 3340 clausal problems in TPTP, (ii) the 1836 problems from the list software reuse application (see [8]). On each problem, Vampire was run with 3 different literal selection functions using the DISCOUNT algorithm, and with the same 3 different selection functions using the LRS-based algorithm. Therefore, altogether we compared the two algorithms on 15,528 tests. To summarize the results, we consider only tests satisfying the following conditions: (i) exactly one of the two algorithms solved the problem, or (ii) both algorithms solved the problem, and at least one of them spent more than 10 seconds on it. Of 1726 benchmarks, 1492 were solved by the LRS-based algorithm, and 1045 by the DISCOUNT algorithm. The DISCOUNT algorithm was not able to solve 681 problems solved by the LRS, compared to 234 problems not solved by the LRS while solved by the DISCOUNT algorithm. The LRS was faster on 241 of the problems solved by both algorithms, while the DISCOUNT algorithm was faster on 161 problems.

We also compared the OTTER algorithm with and without LRS using the same benchmark suites as in the previous section. Of 1318 benchmarks selected according to the same criterion as above, 1267 were solved by the LRS-based algorithm, and 589 by the standard OTTER algorithm.

To illustrate how LRS influences the proof-search statistics in terms of the share of active clauses in the kept clauses, consider the TPTP problem ANA003-1. This problem was solved by no algorithm. The following table summarizes the total number of used and kept clauses.

	DISCOUNT	OTTER	LRS
used	8,191	1,967	42,050
kept	1,473,106	236,389	51,751

The DISCOUNT algorithm could process about 4 times more active clauses than the OTTER algorithm. However, it comes at a price of not performing some simplification steps. Also, the DISCOUNT algorithm kept about 6 times more clauses than the OTTER algorithm since it could not recognize that some of them are redundant w.r.t. passive clauses. The LRS-based algorithm could process about 21 times more active clauses than the OTTER algorithm and about 4 times more than the DISCOUNT algorithm. The small difference between the numbers of the kept and the active clauses shows that the calculations of reachable clauses made by LRS were quite precise.

To compare the LRS with weight-limit based approaches, we experimented with the 75 problems from the CASC-16 competition in the mixed category. We compare the results obtained by Vampire using the following strategies based on the OTTER algorithm: LRS, four different fixed weight limits (50, 40, 30, 20) and Incremental Weight Limit (winc). Only those 51 problems for which a proof was found by at least one strategy are considered. In the table below we give the total number of problems solved by each strategy.

strategy	LRS	50	40	30	20	winc
solved	48	45	36	27	21	33

As it can be seen from the results, the Limited Resource Strategy solves more problems than the Incremental Weight Limit Strategy or any strategy using fixed weight limit, even when the values of the weight limit are optimal for this benchmark suite. There is a problem that could only be solved by LRS (ANA002-4), and for 3 more problems the time obtained by this strategy was considerably better than by any other strategy. LRS also gives better average time than the strategy with weight limit 50, when they are compared on problems solved by both of them.

References

1. J. Avenhaus, J. Denzinger, and M. Fuchs. DISCOUNT: a system for distributed equational deduction. In J. Hsiang, editor, *Proc. RTA-95*, volume 914 of *LNCS*, pages 397–402, 1995.
2. H. de Nivelle. *Bliksem 1.10 User's Manual.* MPI für Informatik, Saarbrücken, 2000.
3. E.L. Lusk. Controlling redundancy in large search spaces: Argonne-style theorem proving through the years. In A. Voronkov, editor, *Proc. LPAR'92.*, volume 624 of *LNAI*, pages 96–106, St.Petersburg, Russia, 1992.
4. W.W. McCune. OTTER 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory, 1994.
5. M. Moser, O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann, and K. Mayr. SETHEO and E-SETHEO—the CADE-13 systems. *Journal of Automated Reasoning*, 18:237–246, 1997.
6. A. Riazanov and A. Voronkov. Vampire. In H. Ganzinger, editor, *Proc. CADE-16*, volume 1632 of *LNAI*, pages 292–296, 1999.
7. S. Schulz. System abstract: E 0.3. In H. Ganzinger, editor, *Proc. CADE-16*, LNAI, pages 297–301, 1999.
8. J. Schumann and B. Fischer. NORA/HAMMR: Making deduction-based software component retrieval practical. In *Proc. ASE-97*, pages 246–254. IEEE Computer Society Press, 1997.
9. G. Sutcliffe. The CADE-16 ATP system competition. *Journal of Automated Reasoning*, 24(3):371–396, 2000.
10. T. Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, 1997.
11. C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, T. Engel, E. Keen, C. Theobalt, and D. Topic. System description: SPASS version 1.0.0. In H. Ganzinger, editor, *Proc. CADE-16*, volume 1632 of *LNAI*, pages 378–382, 1999.

A Verification Approach for Distributed Abstract State Machines

Robert Eschbach

Department of Informatics, University of Kaiserslautern, Germany
e-mail: eschbach@informatik.uni-kl.de

1 Introduction

Distributed ASMs represent a general mathematical model of concurrent computation. In particular its notion of partially ordered runs allows as much concurrency as logically possible. Distributed ASMs have been used successfully to specify and verify distributed algorithms including the Bakery Algorithm of Lamport [BGR95,GR00] and the termination detection algorithm of Dijkstra, Feijen and van Gasteren [Esc99]. Distributed ASMs have also been used to define formal semantics for several programming (specification) languages like C [GH93], and more recently SDL [EGGP00].

The notion of partially ordered run is a general and adequate description of distributed computations. However, in the ASM literature often the use of partially ordered runs is avoided. We believe one reason for this can be found in the more complex structure of partially ordered runs. For example, in a partially ordered run a move is executed in several states. In general there exists no unique global state in which a move is executed. This makes verification somewhat difficult.

In order to overcome these problems and make the handling of partially ordered runs more feasible, the notion of *maximal transition graph* is introduced. A maximal transition graph can be seen as a general description of all possible behaviors and can be constructed in an intuitive way. In a certain sense, the maximal transition graph contains all partially ordered runs. It can be used within the verification process to compare different runs or to reason about a single run. Some central concepts like indisputable terms, pre- and post-states of a move within a partially ordered run (cf. [GR00]) can be easily found in the maximal transition graph. We believe, that the use of these kind of graphs eases the process of verification. The concept of maximal transition graphs is illustrated by some examples.

2 Maximal Transition Graphs

In this section we illustrate the notion of *maximal transition graph*. We presume the reader to be familiar with [Gur95]. We start with the deterministic case. In the following let $\mathcal{A}$ be a distributed deterministic ASM. Furthermore we assume only infinite runs. The maximal transition graph associated with $\mathcal{A}$ represent all possible behaviors on states. It can be used to analyze partially ordered runs. Due to its intuitive representation as a graph, it can be used within the verification of properties for partially ordered runs. Each partially run can be found within the maximal transition graph. In this way a comparison of partially ordered runs becomes more feasible.

Starting from the initial states all possible next states are related by an edge labeled with the corresponding executing agent. In order to avoid cycles between states we use a ranking function which excludes edges from higher to lower ranked states. Initial states have rank 0. Their successor states rank 1, and so on. We start with an example.

Example 1. We consider a distributed ASM $\mathcal{A}$ with a static set of internal agents a, b. Agent a increments x, agent b increments y, where x, y denotes constants of type NAT, interpreted within states as the set of natural numbers. Initially $x = y = 0$ holds. There are no external agents. The programs of both a and b possess the 'empty enabling condition' denoted by {Enable :}.

```
a: {Enable: } x := x + 1
b: {Enable: } y := y + 1
```

In figure 1 (for simplicity idle steps and ranking are not shown) one can see parts of the maximal transition graph associated with ASM $\mathcal{A}$. Consider the execution path (b, b, a) which transforms the initial state $(0,0) = (x, y)$ into state $(1,2)$ and correspond to the partially ordered moves (1) depicted in figure 1. The first move of b is

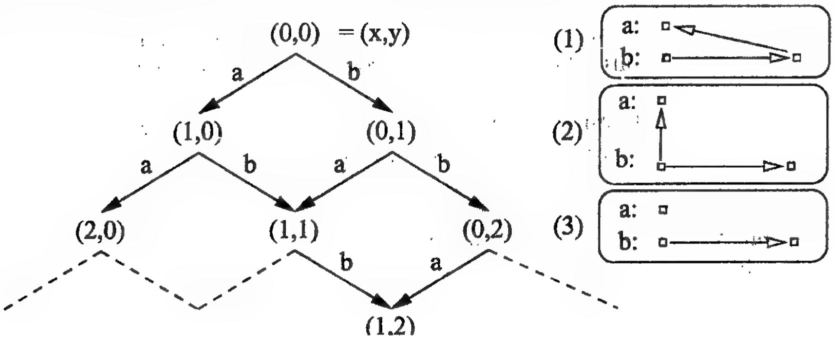


Fig. 1. Maximal Transition Graph

smaller than the second move of b (presented by an arrow relating the first and the second move of b) which is in turn smaller than the first move of a.

Now consider the execution paths $\{(b,b,a),(b,a,b)\}$ which correspond to the partially ordered set of moves (2) depicted in figure 1. Intuitively, the second move of b and the first move of a can be executed in any order (they are independent) after b has performed its first move. This can be seen in (2) where the second move of b and the first move of a are incomparable. Note that (1) is one linearization of (2).

In figure 1 one can see that the first move of b and the first move of a may be independent, too. Consider the execution paths $\{(a,b,b),(b,b,a),(b,a,b)\}$ and the corresponding partially ordered set of moves (3) which additionally expresses this independence. □

2.1 Maximal Independent Runs

Each maximal path starting from an initial state can be interpreted as a linear run. The structural information of the maximal transition graph can be used to obtain more independent versions of this run. As illustrated in the example above one can enlarge a linear path in the following way: each state on the run which has more than one incoming edge can be used for an enlargement by tracing back one or more of those edges and predecessors until an already 'visited' node is reached. For example, tracing back the edge from $(1,1)$ to $(1,2)$ and the edge from $(0,1)$ to $(1,1)$ leads to the partially ordered set of moves (2) depicted in figure 1. From the construction of the maximal transition graph one easily obtains in this way a maximal independent version of a linear run.

2.2 Pre-/Post-States

The maximal transition graph can also be used to determine the set of pre- and post-states associated with a move t. In the example 1 above the set of pre- and post-states of the second move of b w.r.t. the partial ordered set of moves (3) depicted in figure 1 can be easily found within the maximal transition graph: pre-states of this move are related to the transitions $(0,1)$ to $(0,2)$ and $(1,1)$ to $(1,2)$. The pre-states are $\{(0,1),(1,1)\}$ whereas the post-states are $\{(0,2),\ (1,2)\}$.

2.3 Indisputable Locations

Whenever a location has the same content in all pre-states of a move we say that this location is *indisputable* for this move (cf. [GR00]). In the example 1 the location x (more precisely $(x,())$) is indisputable for all moves of a in in all partially ordered set of moves (1), (2), and (3). On the other side, the location y is indisputable for all moves of b in in all partially ordered set of moves. This can be seen directly within the programs associated with a and b. For example, the location x is completely under control of a and its content in a state completely determined by the a-predecessors within each partially ordered set of moves (1), (2), and (3).

Example 2 (Example 1 continued). We change the example 1 slightly in the following way:

```
a: {Enable: }        x := x + 1
b: {Enable: even(x)} y := y + 1
```

The program of agent b has changed. It now makes use of the predicate even which characterizes the even natural numbers. Moves of agent b are enabled only if the content of location x in all pre-states denotes an even number.

In this example the location y is still completely under control of b. But now the contents is also dependent of a-moves. This changes the indisputable portions of states. This can be directly seen in the corresponding maximal transition graph. □

Remark 1 (Non-Deterministic Runs). The tracing back construction for maximal transition graphs can be extended to non-deterministic runs. We forego a precise description of this construction in this version of the paper. □

References

[BGR95] Egon Börger, Yuri Gurevich, and Dean Rosenzweig. The bakery algorithm: Yet another specification and verification. In E. Börger, editor, *Specification and Validation Methods*, pages 231–243. Oxford University Press, 1995.

[EGGP00] Robert Eschbach, Uwe Glässer, Reinhard Gotzhein, and Andreas Prinz. On the Formal Semantics of SDL-2000: A Compilation Approach Based on an Abstract SDL Machine. In Y. Gurevich, P.W. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines - Theory and Applications*, number 1912 in LNCS. Springer, 2000.

[Esc99] Robert Eschbach. A termination detection algorithm: Specification and verification. In Jeanette M. Wing, Jim Woodcock, and Jim Davies, editors, *Proc. of FM'99 - World Congress on Formal Methods in the Development of Computing Systems*, number 1709 in LNCS, pages 1720–1737, 1999.

[GH93] Yuri Gurevich and James K. Huggins. The semantics of the c programming language. In *Selected papers from CSL'92 (Computer Science Logic)*, LNCS, pages 274–308. Springer, 1993.

[GR00] Yuri Gurevich and Dean Rosenzweig. Partially ordered runs: A case study. In Yuri Gurevich, Philipp W. Kutter, Martin Odersky, and Lothar Thiele, editors, *Abstract State Machines: Theory and Applications, Proc. of International Workshop, ASM2000, Monte Verità, Switzerland*, number 1912 in LNCS, pages 131–150. Springer, March 2000.

[Gur95] Yuri Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford Univ. Press, 1995.

Program Transformation and Synthesis

Transformational Construction of Correct Pointer Algorithms

Thorsten Ehm

Institut für Informatik
Universität Augsburg
e-mail: Ehm@Informatik.Uni-Augsburg.DE

1 Introduction

Algorithms on pointer structures are often used in lower levels of implementation. Although in modern programming languages (e.g. in Java) they are hidden from the programmer, they play a significant rôle at the implementation level due to their performance. But this advantage is bought at high expense. Pointer algorithms are very error-prone and so there is a strong demand for a formal treatment and development process for pointer algorithms. There are some approaches to achieve this goal:

Several methods [2, 10, 11] use the wp-calculus to show the correctness of pointer algorithms. There only properties of the algorithms are proved but the algorithms are not derived from a specification. So the developer has to provide an implementation. In these approaches proving trivialities may last several pages. Butler [7] investigates how to generate imperative procedures from applicative functions on abstract trees. To achieve this he enriches the trees by paths to eliminate recursion. A recent paper by Bornat [5] shows that it is possible, but difficult to reason in Hoare logic about programs that modify data structures defined by pointers. Reynolds [16] also uses Hoare logic and tries to improve a method described in a former paper of Burstall [6] to show the correctness of imperative programs that alter linked data structures.

In [13] Möller proposed a framework based on relation algebra to derive pointer algorithms from a functional specification. He shows that the rules presented also are capable of handling more difficult multi-linked data structures like doubly-linked lists or trees. However the derived algorithms are still recursive. Our goal is to improve this method by showing how to derive imperative algorithms and so achieve a more complete calculus for transformational derivation of pointer algorithms. Based on the method by Möller a recent paper by Richard Bird [4] shows how one can derive the Schorr-Waite marking algorithm in a totally functional way.

This paper shows how to use the transformation of Paterson and Hewitt (P & H) to derive imperative pointer algorithms. To achieve this we take the recursive pointer algorithms derived from functional descriptions using the method of Möller. These are transformed via the P & H transformation scheme into an imperative version. Despite the inefficient general runtime performance of the scheme that results from P & H, we get well performing algorithms. As a side effect the amount of selective updates in memory is improved by eliminating ineffective updates that are only used to pass through the pointer structure. This is not a trivial task, because in general it is not decidable if an update really changes links of the pointer structure. Some systems do such optimization during runtime but not in such an early state of software development.

We will show how these aims can be achieved for a class of pointer algorithms that first pass through a pointer structure to find the position where they have to do some proper changes. A similar transformation scheme for a class of algorithms that not only alter but also delete links is in preparation. Be aware that we are not interested in algorithms that do not alter the link structure but only the contents of the nodes (like for example `map`). The advantage of the presented method over the previously mentioned approaches using wp-calculus or Hoare logic are apparent. All these methods provide correct algorithms. Though the presented one treats a class of functions whereas the other methods have to be applied on every new algorithm. You also do not have to provide an implementation for a specification which is a time-consuming task. Not least, the transformational approach is more likely to be the easier one to automate.

2 Pointer Structures and Operations

To make this abstract self-contained as far as possible we present a short introduction to pointer structures and how they are used in [13].

In our model a pointer structure $\mathcal{P} = (s, P)$ consists of a store P and a list of entries s. The entries of a pointer structure are addresses $\mathcal{A}$ that form starting points of the modeled data structures. We assume a distinguished element $\diamond \in \mathcal{A}$ representing a terminal node (e.g. null in C or nil in Pascal). A store is a family of relations (more precisely partial maps) either between addresses or from addresses to node values $\mathcal{N}_j$ such as *Integer* or *Boolean*. Each relation represents a selector on the records like e.g. *head* and *tail* for lists with functionality $\mathcal{A} \to \mathcal{N}_j$ respectively $\mathcal{A} \to \mathcal{A}$.

Each abstract object implemented is represented by a pointer structure (n, P) with a single entry $n \in \mathcal{A}$ which represents the entry point of the data structure such as for example the root node in a tree. For convenience we introduce the access functions

$$ptr(n, L) = n \quad \text{and} \quad sto(n, L) = L$$

We want to give only the necessary definitions of operations used in this paper. More of them and proofs can be found in [13]. The following operations on relations all are canonically lifted to families of relations. Algorithms on pointer structures stand out for altering links between elements. Such modification has to be modeled in the calculus as well. We use an update operator | (pronounced "onto") that overwrites relation S by relation R:

Definition 1. $R \mid S \overset{\text{def}}{=} R \cup \overline{dom(R)} \bowtie S$

Here we have used the *domain restriction* operator $\bowtie$ which is defined as $L \bowtie S = S \cap (L \times N)$ to select a particular part of $S \subseteq \mathcal{P}(M \times N)$. The update operator takes all links defined in R and adds the ones from S that no link starts from in R. To be able to change exactly one pointer in one explicit selector we define a sort of a "mini-store" that is a family of partial maps defined by:

Definition 2. $(x \overset{k}{\to} y) \overset{\text{def}}{=} \left\{ \begin{matrix} \{(x,y)\} & \textit{for selector } k \\ \emptyset & \textit{otherwise} \end{matrix} \right\}$

It is clear that overwriting a pointer structure with links already defined in it does not change the structure:

Lemma 1. $S \subseteq T \;\Rightarrow\; S \mid T = T$ *(Annihilation)*

To have a more intuitive notation leaned on traditional programming languages, we introduce the following selective update notation:

Definition 3. *For selector k of type $\mathcal{A} \to \mathcal{A}$*
$(n, P).k := (m, Q) \overset{\text{def}}{=} (n, (n \overset{k}{\to} m) \mid Q)$

which overwrites Q with a single link from n to m at selector k. Selection is done the same way:

Definition 4.
k of type $\mathcal{A} \to \mathcal{A}$: $(n, P).k \overset{\text{def}}{=} (P_k(n), P)$
k of type $\mathcal{A} \to \mathcal{N}_j$: $(n, P).k \overset{\text{def}}{=} P_k(n)$

To have the possibility to insert new (unused) addresses into the data structure we define the newrec operator. Let k range over all selectors used in the modelled data structure. Then the operator newrec$(L, k : x_k)$ alters the store L to have a new record previously not in L and each selector k pointing to x_k. So for example newrec$(L, head : 3, tail : \diamond)$ returns a pointer structure (m, K) with m a new address previously not used in L and store K consisting of L united with two new links $(m \overset{head}{\to} 3)$ and $(m \overset{tail}{\to} \diamond)$. If it is clear from the context which selectors are used we only enumerate the respective components. So the previous expression becomes newrec$(L, \langle 3, \diamond \rangle)$.

3 A Running Example and the Problem

In this section we want to use a functional description of list concatenation. This function serves as our running example during the derivation of the transformation scheme. We will use Haskell [3] notation to denote functional algorithms:

```
cat [] ys       = ys
cat (x:xs)  ys = x : cat xs ys
```

We assume that the two lists are acyclic and do not share any parts. So the following pointer algorithm can be derived by transformation using the method of [13]:

$$\begin{aligned} cat_p(m,n,L) = \text{if } m \neq \diamond \ &\text{then } (m,L).tail := cat_p(L_{tail}(m),n,L) \\ &\text{else } (n,L) \end{aligned}$$

The two pointer structures (m, L) and (n, L) are representations of the two lists. Addresses m and n model the starting points, whereas L is the memory going with them. In other words m and n form links to the beginning of two lists in memory L.

Note that this is only one candidate of possible imnlementations for the functionally described specification of cat. Because we are interested in algorithms performing minimal **destructive** updates we did not derive a persistent variant such as the standard, partially copying interpretation in functional languages. Although that would also be possible.

We now have a linear recursive function working on pointer structures. But what we want is an imperative program that does not use recursion. By investigating the execution order of cat_p we can see, that cat_p calculates a term of the following form:

$$(m, L).tail := ((L_{tail}(m), L).tail := ...(n, L))$$

If you remember the definition of the := operator, this means that updates are performed from right to left.

$$(m \overset{tail}{\to} L_{tail}(m)) \mid (\ldots \mid ((L^k_{tail}(m) \overset{tail}{\to} n) \mid L) \ldots)$$

This shows that the derived algorithm uses the update operator not only to properly alter links but also to just pass through the structure while returning from the recursion.

As we can see, there are several such updates that do not alter the pointer structure. For example $(m \overset{tail}{\to} L_{tail}(m))$ is already contained in L and does not change the pointer structure $(\ldots \mid ((L^k_{tail}(m) \overset{tail}{\to} n) \mid L) \ldots)$ if the previous updates do not affect this part of L. This is the case for several algorithms on pointer linked data structures, because most of them first have to scan the structure to find the position where they have to do the proper changes.

We now define the following abbreviations to get a standardized form for later transformations:

$$\begin{array}{lll} K(m,n,L) \overset{\text{def}}{=} (L_{tail}(m), n, L) & B(m,n,L) \overset{\text{def}}{=} m \neq \diamond & \phi_k(u,v) \overset{\text{def}}{=} v.k := u \\ H(m,n,L) \overset{\text{def}}{=} (n, L) & E(m,n,L) \overset{\text{def}}{=} (m, L) & \end{array}$$

Abbreviating (m, n, L) to x the derived pointer algorithm can then be written as

$$\begin{aligned} cat_p(x) = \text{if } B(x) \ &\text{then } \phi_{tail}(cat_p(K(x)), E(x)) \\ &\text{else } H(x) \end{aligned}$$

4 From Linear via Tail Recursion to While Programs

In transformational program design the transformation of a linearly recursive function to an imperative version always has two steps: First transform the linear recursion into tail recursion. Then apply a transformation scheme [15] like the following to get a while program:

```
$f(x) = $ if $B(x)$ then $f(K(x))$
               else $H(x)$
--------------------------------- $\updownarrow$ ---------------------------------
$f(x) = $ var $vx := x$;
        while $B(vx)$ do $vx := K(vx)$;
        $H(vx)$;
```

But cat_p does not have tail recursive form. So we first have to find a way to transform cat_p into the right form. There are several schemes to derive a tail recursive variant from a linear recursive function [1]. One of the most popular is to change the evaluation order of parentheses in the calculated expression. To be able to do this one needs a function ψ that fulfills the equation $\phi(\phi(r,s),t) = \phi(r,\psi(s,t))$. To find such a ψ is possible only in very rare cases of ϕ. One of these is that ϕ is associative. In this case you can choose $\psi = \phi$. An other — similar — case is to change the order of operands. Here it is necessary that $\phi(\phi(r,s),t) = \phi(\phi(r,t),s)$ or more generally you need a ψ with $\phi(\psi(r,s),t) = \psi(\phi(r,t),s)$. The previously described rules assume that ϕ is good-natured enough to satisfy one of the properties mentioned. However, our function $\phi_{tail}(u,v)$ in cat_p does not show any of these properties. Another transformation uses function inversion to calculate the parameter values from the results. Here one only has to find an inverse $\overline{K}$ of K. But the function $K(m,n,L) \overset{\text{def}}{=} (L_{tail}(m),n,L)$ in general is not invertible. So is there no way to get a tail recursive version of cat_p ?

5 The Transformation Scheme of Paterson/Hewitt

In 1970 Paterson and Hewitt presented a transformation scheme that makes it possible to transform any linear recursive function to a tail recursive one [1]. This rule normally is only of theoretical interest because of the bad runtime performance of the resulting function. P & H applied the idea of the method mentioned in Section 4 using the inverse function $\overline{K}$ to make the step from K^{i+1} to K^i, but exhaustively recalculated K^i from the start. The evolving scheme is:

$$\begin{array}{|l|}
\hline
F(x) = \text{if } B(x) \ \text{ then } \phi(F(K(x)), E(x)) \\
\qquad\qquad\quad \text{else } H(x) \\
\hline
\qquad\qquad\qquad\qquad \updownarrow \qquad\qquad\qquad\qquad\qquad\qquad [\text{ P \& H} \\
\hline
F(x) = G(n0, H(m0)) \text{ where} \\
\qquad (m0, n0) = num(x, 0) \\
\qquad num(y,i) = \text{if } B(y) \ \text{ then } num(K(y), i+1) \\
\qquad\qquad\qquad\qquad\qquad\quad \text{else } (y,i) \\
\qquad it(y,i) = \text{if } i \neq 0 \ \text{ then } it(K(y), i-1) \\
\qquad\qquad\qquad\qquad\quad \text{else } y \\
\qquad G(i,z) = \text{if } i \neq 0 \ \text{ then } G(i-1, \phi(z, E(it(x, i-1)))) \\
\qquad\qquad\qquad\qquad\quad \text{else } z \\
\hline
\end{array}$$

The function num calculates the number of iterations that have to be done until the termination condition is fulfilled as well as the final value. These values are used by function G to change the evaluation order of the calculated term. For this, G uses the function it to iterate K to achieve the inverse $\overline{K}$ of K by doing one iteration less than had to be done for K. So G can start with the calculations done in the deepest recursion step first and then ascend from there using the inverse of K.

As we have seen, function it is only used to calculate the powers of K and we have $it(y,i) = K^i(y)$, so we can abbreviate $\phi(z, E(it(x,i-1)))$ to $\phi(z, E(K^{i-1}(x)))$. This certainly is only a cosmetic change, because K^{i-1} has to be calculated exactly the same way it is in the original transformation scheme. But this gives the basis to future simplification, because $K^{i-1}(x)$ is only used as parameter for E and will be eliminated in further steps.

6 Deriving a General Transformation Scheme

We now present an application of the P & H transformation scheme to pointer algorithms using the function ϕ_k to pass through a pointer data structure.

By investigation of function $\phi_k((m,L),(n,L)) = (n,(n \xrightarrow{k} m) \mid L)$ we can see that ϕ_k updates the link starting from m via selector k and simultaneously sets n as the new starting entry of the resulting pointer structure. It is apparent that such a restricted function can not provide the simplification we aim to achieve, namely elimination of effect-less updates. So we use the technique of generalization and introduce a more flexible function $\psi_k(l,m,(n,L)) = (l,(m \xrightarrow{k} n) \mid L)$ that handles the altered address and the resulting entry independently. With this function we are in the position to eliminate the quasi-updates that do not alter the structure but are only used for passing through the pointer structure. One can say that ψ_k "eats up" the effect-less updates of ϕ_k:

Lemma 2. *If* $(t \xrightarrow{k} v) \subseteq (v \xrightarrow{k} u) \mid U$ *then for all* s

$$\psi_k(s, t, \phi_k((u, U), (v, U))) = \psi_k(s, v, (u, U))$$

Now we return to the P & H transformation scheme. There the function G applies ϕ_k so that this lemma can be used to simplify G. We apply the lemma to all instances of G that only pass through the pointer structure. This means as long as the condition B is fulfilled we apply Lemma 2 and eliminate one application of ϕ_k. So the precondition of Lemma 2 has to hold for all those cases.

Lemma 3. *We abbreviate* $p^{(i)} \overset{\text{def}}{=} ptr(E(K^i(x)))$. *Then under the condition*

$$\forall i \in \{0, \ldots, n0\} : ptr(z) = p^{(i)} = p^{(i-1)} \vee (p^{(i)} \neq p^{(i-1)} \wedge (p^{(i-1)}, p^{(i)}) \in sto(z))$$

we can simplify $G(i, z)$ *to:*

$$\begin{aligned} G(i,z) = \text{if } i \neq 0 \ \text{ then } & \psi_k(ptr(E(x)), ptr(E(K^{i-1}(x))), z) \\ \text{else } & z \end{aligned}$$

Remembering that K is the function performing the run through the pointer structure we can express the condition in human-understandable form. The pair $(p^{(i-1)}, p^{(i)})$ consists of the values under function E of two such successive elements that come from the pass-through via K. Now, either these are equal which means the links form a cycle and the simplification is trivial. Or they are not equal and the memory already contains the pair. Then an update using these values will not change anything and can be eliminated.

With $n0 = \min\{j : \neg B(K^j(x))\}$ this is a condition that in some cases can not be checked easily. But normally one proves a more general assertion. For function *cat* for example we have acylic lists and we can show that the condition holds for all successive pairs of elements in the list.

Now that G is not recursive anymore we can instantiate the application of G with its actual parameters. The test $i \neq 0$ is only calculated once. By inspection of *num* that calculates $n0$ (the actual argument for parameter i) we see that the inequality test can be done without $n0$:

Lemma 4. $n0 \neq 0 \Leftrightarrow B(x)$

So the scheme of Paterson and Hewitt simplifies to

$$\begin{aligned} F(x) = \text{if } B(x) \ \text{ then } & \psi_k(ptr(E(x)), ptr(E(K^{n0-1}(x))), H(m0)) \\ \text{else } & H(m0) \\ \text{where } \ & (m0, n0) = num(x, 0) \\ & num(y, i) = \text{if } B(y) \ \text{ then } num(K(y), i+1) \\ & \qquad\qquad\qquad\quad \text{else } (y, i) \end{aligned}$$

A straightforward induction shows that $m0 = K^{n0}(x)$. So the calculation of $m0$ can be done simultaneously with the calculation of K^{n0-1}. This is achieved by a slightly changed pair of functions num' and num'' that replace *num*. For this we extend the domain of K by a special element $\triangle$ with $K(\triangle) = y$ that models the imaginary predecessor of y under K:

$$\begin{aligned} num'(y) &= \text{if } B(y) \ \text{ then } num''(y) \\ &\qquad\qquad\quad \text{else } \triangle \\ num''(y) &= \text{if } B(K(y)) \ \text{ then } num''(K(y)) \\ &\qquad\qquad\qquad\quad \text{else } y \end{aligned}$$

and obtain

Lemma 5. $num'(x) = K^{n0-1}(x)$ *and thus also* $m0 = K(num'(x))$

This is the basis for the following transformation:

```
$F(x) = $ if $B(x)$ then $\psi_k(ptr(E(x)), ptr(E(k0)), H(K(k0)))$
               else $H(K(k0))$
         where $k0 = num'(x)$
               $num'(y) = $ if $B(y)$ then $num''(y)$
                                      else $\triangle$
               $num''(y) = $ if $B(K(y))$ then $num''(K(y))$
                                          else $y$
──────────────────────── ↕ ──────────────────────── [ unfold def. of $num'$ and $k0$
$F(x) = $ if $B(x)$ then $\psi_k(ptr(E(x)), ptr(E(k0)), H(K(k0)))$
               else $H(K($if $B(x)$ then $num''(x)$ else $\triangle))$
         where $k0 = $ if $B(x)$ then $num''(x)$
                                 else $\triangle$
               $num''(y) = $ if $B(K(y))$ then $num''(K(y))$
                                          else $y$
```

Although there is a term $E(k0)$ in the scheme the case that $E(\triangle)$ has to be evaluated can never be reached. So there is no need to define $E(\triangle)$. Now num'' is the only recursive function; it is even tail-recursive so that we are in the position to use the transformation scheme presented in Section 4 to achieve an imperative while program:

```
$F(x) = $ if $B(x)$ then $\psi_k(ptr(E(x)), ptr(E(k0)), H(K(k0)))$
               else $H(x)$
         where $k0 = $ if $B(x)$ then var $vx := x$
                                      while $B(K(vx))$ do $vx := K(vx)$
                                      $vx$
                                 else $\triangle$
──────────────────────────── ↕ ──────────────────────────── [ Simplification
$F(x) = $ var $vx := x$
          if $B(x)$ then while $B(K(vx))$ do $vx := K(vx)$
                         $\psi_k(ptr(E(x)), ptr(E(vx)), H(K(vx)))$
               else $H(x)$
```

The scheme that has evolved from our calculations now is:

```
$F(x) = $ if $B(x)$ then $\phi(F(K(x)), E(x))$
               else $H(x)$
──────────────────────── ↕ ──────────────────────── [ Conditions of Lemma 3
$F(x) = $ var $vx := x$
          if $B(x)$ then while $B(K(vx))$ do $vx := K(vx)$
                         $\psi_k(ptr(E(x)), ptr(E(vx)), H(K(vx)))$
               else $H(x)$
```

To return to our example in the previous sections we now can transform the recursive version of cat_p to an iterative program by using the derived scheme. First we check the applicability condition of our scheme abbreviating $T_i = L^i_{tail}(m)$:

$$\forall i \in \{0, \dots, \min\{j : T_j = \diamond\}\} : n = T_i = T_{i-1} \vee (T_i \neq T_{i-1} \wedge (T_{i-1}, T_i) \in L)$$

The first disjunct is not fulfilled by the assumption that the two lists do not share any parts. But the second disjunct is true by acyclicity of p. So some simplification leads us to the imperative algorithm one has in mind:

```
$cat_p(m, n, L) = $ var $vm := m$
                  if $m \neq \diamond$ then while $L_{tail}(vm) \neq \diamond$ do $vm := L_{tail}(vm)$
                                       $(m, (vm \overset{tail}{\rightarrow} n) \mid L)$
                                  else $(n, L)$
```

7 Further Applications

In this section we want to show that the developed scheme is applicable to several algorithms passing through a pointer-linked data structure.

7.1 Insertion into a Sorted List

In [8] several algorithms on lists are derived with the calculus presented in [13]. We choose insertion into a sorted list as a first example. The function `insert` is defined like this:

```
insert x []     = [x]
insert x (y:ys) = if x ≤ y then x:(y:ys)
                           else y:(insert x ys)
```

We can bring the derived pointer algorithm $insert_p$ into the form needed by our scheme.

```
$insert_p(m, n, L) =$
  if $n \neq \diamond \wedge L_{val}(m) > L_{head}(n)$ then $q.tail := insert_p(m, L_{tail}(n), L)$
                                    else newrec$(L, \langle L_{val}(m), (n, L)\rangle)$
```

Now we can apply the scheme and after a simplification step achieve an imperative algorithm for insertion into a sorted list:

```
$insert_p(m, n, L) =$ var $vn = n$
                  if $(n \neq \diamond \wedge L_{val}(m) > L_{head}(n))$ then
                    while $(L_{tail}(vn) \neq \diamond \wedge L_{val}(m) > L_{head}(vn))$ do $vn = L_{tail}(vn)$
                    $\psi_{tail}(n, vn,$ newrec$(L, \langle L_{val}(m), (vn, L)\rangle))$
                  else newrec$(L, \langle L_{val}(m), (n, L)\rangle)$
```

7.2 Insertion into a Tree

To show an example using a data structure different from lists we show how insertion into a tree can be derived from our scheme. It is nearly as easy as the other examples. We use the algorithm derived in [13] from the following functional specification:

```
ins x Empty        = Tree(Empty,x,Empty)
ins x Tree(l,y,r) = if x ≤ y then Tree(ins x l,y,r)
                             else Tree(l,y,ins x r)
```

We abbreviate $\widetilde{x} = L_{val}(x)$ and $p = (m, L)$ as before and get:

```
$ins_p\ x\ p =$ if $m = \diamond$ then newrec$(L, \langle\diamond, x, \diamond\rangle)$
                else if $x \leq \widetilde{m}$ then $p.l := ins_p\ x\ p.l$
                                 else $p.r := ins_p\ x\ p.r$
```

The algorithm can be transformed into the form needed by our scheme with the help of the conditional operator _ ? _ : _ as used in several programming languages.

```
$ins_p\ x\ p =$ if $m \neq \diamond$ then $\phi_{(x \leq \widetilde{m}?l:r)}(ins_p\ x\ (x \leq \widetilde{m}?p.l : p.r)), p)$
                else newrec$(L, \langle\diamond, x, \diamond\rangle))$
```

Now we can use our scheme to achieve an imperative algorithm for insertion into a tree.

```
$ins_p\ x\ (m, L) =$ var $vm = m$
                if $m \neq \diamond$ then while $(h := x \leq \widetilde{vm}?L_l(vm) : L_r(vm)) \neq \diamond$ do $vm := h$
                                $\psi_{(x \leq \widetilde{vm}?l:r)}(m, vm,$ newrec$(L, \langle\diamond, x, \diamond\rangle)$
                else newrec$(L, \langle\diamond, x, \diamond\rangle)$
```

Here we have used an assignment inside the condition of the while loop. Otherwise the algorithm would have to use the conditional operator _ ? _ : _ twice or introduce two new while loops. But we do not think this would make the algorithm more readable.

8 Conclusion

We have shown how the transformation of Paterson and Hewitt can be used to achieve imperative algorithms on pointer-linked data structures. Although the transformation of Paterson and Hewitt normally is only of theoretical interest because of its very bad runtime behaviour, well-performing algorithms are derived. This leads to a general methodology for the derivation of pointer algorithms.

At the example algorithm for insertion into a tree it can be seen, that there is a need for more sophisticated schemes based on the presented one. It also seems possible that algorithms changing more than one link such as deletion from a list can be treated the same way. For this, one have to divide the job into several parts altering only one link, applying the scheme and afterwards putting the parts together.

Further research will investigate this and other starting points to complete the methodology. Also a (semi-) automatic system checking the side-conditions and so supporting the developer of such algorithms is in work.

References

1. F.L.Bauer, Wössner: *Algorithmische Sprache und Programmentwicklung*, in German, Springer, Berlin, 1981
2. A. Bijlsma: *Calculating with pointers.* Science of Computer Programming **12**, Elsevier 1989, 191–205
3. R. Bird: *Introduction to Functional Programming using Haskell*, 2nd edition, Prentice Hall Press, 1998
4. R. Bird: *Unfolding Pointer Algoritms*, under consideration for publication in Journal of Functional Programming, available from: `http://www.comlab.ox.ac.uk/oucl/work/richard.bird/publications`
5. R.Bornat: *Proving pointer programs in Hoare logic.* Proceedings of MPC 2000, Ponte de Lima , LNCS 1837, Springer 2000, 102–126
6. R. Burstall: *Some techniques for proving correctness of programs which alter data structures.* In B. Meltzer and D. Michie eds, Machine intelligence 7, Edinburgh University Press, 1972, 23–50
7. M. Butler: *Calculational derivation of pointer algorithms from tree operations.* Science of Computer Programming **33**, Elsevier 1999, 221–260
8. T. Ehm: *Case studies for the derivation of pointer algorithms.* to appear
9. C. A. R. Hoare: *Proofs of correctness of data representations.* Acta Informatica **1**, 1972, 271–281
10. J.M. Morris: *A general axiom of assignment.* Theoretical Foundations of Programming Methodology, NATO Advanced Study Institutes Series C Mathematical and Physical Sciences **91**, Dordrecht, Reidel, 1981, 25–34
11. J.M. Morris: *Assignment and linked data structures.* Theoretical Foundations of Programming Methodology, NATO Advanced Study Institutes Series C Mathematical and Physical Sciences **91**, Dordrecht, Reidel, 1981, 35–51
12. B. Möller: *Towards pointer algebra.* Science of Computer Programming **21**, Elsevier, 1993, 57–90
13. B. Möller: *Calculating with pointer structures.* In: R. Bird, L. Meertens (eds.): Algorithmic languages and calculi. Proc. IFIP WG2.1 Working conference, Le Bischenberg. Chapman & Hall 1997, 24–48
14. B. Möller: *Calculating with acyclic and cyclic lists.* In A. Jaoua, G. Schmidt (eds.): Relational Methods in Computer Science. Int. Seminar on Relational Methods in Computer Science, Jan 6–10, 1997 in Hammamet. Information Sciences — An International Journal **119**, 1999, 135–154
15. H. Partsch: *Specification and transformation of programs. A formal approach to software development.* Monographs in Computer Science. Springer, 1990
16. J.C.Reynolds: *Intuitionistic reasoning about shared mutable data structures.* In: Millennial Perspectives in Computer Science, Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare, Palgrave, 2000

A Theoretical Foundation of Program Synthesis by Equivalent Transformation

Kiyoshi Akama[1], Hidekatsu Koike[2], and Hiroshi Mabuchi[3]

[1] Hokkaido University, Kita 11, Nishi 5, Kita-ku, Sapporo, 060-0811, Japan,
e-mail: akama@cims.hokudai.ac.jp,

[2] Hokkaido University, Kita 11, Nishi 5, Kita-ku, Sapporo, 060-0811, Japan,
e-mail: koke@cims.hokudai.ac.jp,

[3] Iwate Prefectural University, 152-52 Sugo Takizawa Iwate 020-0173 Japan,
e-mail: mabu@soft.iwate-pu.ac.jp

Abstract. Equivalent transformation is useful for synthesis and transformation of programs. However, it is not so clear what semantics should be preserved in synthesis and transformation of programs in logic and functional programming, which come from the disagreement of computation models (inference or evaluation) from equivalent transformation. Therefore, we adopt a new computation model, called equivalent transformation model, where equivalent transformation is used not only for program synthesis, but also for computation. We develop a simple and general foundation for computation and program synthesis, and prove the correctness of program synthesis by equivalent transformation.

1 Introduction

Equivalent transformation is one of the most important methods for program synthesis and transformation [4]. For instance, in logic programming [3], a predicate definition consisting of first order formulas or definite clauses is transformed equivalently into a (more efficient) logic program (i.e., a set of definite clauses) by using unfolding, folding, goal replacement, and other transformations. In functional programming, a function definition is transformed equivalently into a (more efficient) functional program by using unfolding, folding, tupling, and other transformations.

In this paper we develop a theoretical foundation of program synthesis by equivalent transformation (ET), define basic concepts, and prove correctness of **ET-based program synthesis**. This theory should contain the following items:

1. Definition of specification, computation, and programs,
2. Definition of correctness of computation and programs (with respect to a specification),
3. Relation between computation and equivalent transformation,
4. Proof of correctness of programs obtained by equivalent transformatiun.

It should be noted that such a theory has not been fully established in the existing theories of logic or functional programming. For instance, computation is regarded not as equivalent transformation but as logical inference (resolution) in logic programming and it is not clear which declarative semantics should be preserved in equivalent transformation for correct program synthesis.

Instead of developing a theory in the existing frameworks of logic or functional computation model, we adopt a new computation model, called **equivalent transformation model** [1]. In the equivalent transformation model, equivalent transformation is used not only for program synthesis but also for computation. This enables us to make a simple and general foundation for computation and program synthesis.

2 Theory of Computation

2.1 Problem Formalization based on Meaning

A **representation system** is a triple $\langle Des, v, Meg\rangle$ of two sets, Des and Meg, and a mapping v from Des to Meg. Each element of Des and Meg are called a **description** and a **meaning**, respectively. A relation $\sim$ on Des is defined by

$$des_1 \sim des_2 \iff v(des_1) = v(des_2).$$

Obviously $\sim$ is an equivalence relation.

A problem on a representation system $\langle Des, v, Meg\rangle$ is a triple $\alpha = \langle des, \pi, \mathcal{K}\rangle$, where des is a description in Des, $\mathcal{K}$ is a set, and π is a mapping from Meg to $\mathcal{K}$. The problem $\alpha = \langle des, \pi, \mathcal{K}\rangle$ on a representation system $\langle Des, v, Meg\rangle$ requires to find the element k in $\mathcal{K}$ determined by $k := \pi(v(des))$.

2.2 Transformation by Rewriting Rules

A **rewriting rule** on Des is defined as a subset of $Des \times Des$. A description des_1 is rewritten into a description des_2 by a rewriting rule r, denoted by $des_1 \xrightarrow{r} des_2$, iff $(des_1, des_2) \in r$. Let Rw be a set of rewriting rules. A description des_1 is rewritten into a description des_2 by Rw, denoted by $des_1 \xrightarrow{Rw} des_2$, iff there is a rewriting rule r in Rw such that $des_1 \xrightarrow{r} des_2$.

One way to solve Problem $\alpha = \langle des, \pi, \mathcal{K} \rangle$ on a representation system $\langle Des, v, Meg \rangle$ is shown.

Algorithm A(Rw)
Let Rw be a set of rewriting rules on Des.
Input: a problem $\alpha = \langle des, \pi, \mathcal{K} \rangle$.
Output: an element in $\mathcal{K}$.

1. Assume that a problem $\alpha = \langle des, \pi, \mathcal{K} \rangle$ on $\langle Des, v, Meg \rangle$ is given.
2. Transform des in Des by repeated application of rewriting rules in Rw into des' in Des.

$$des = des_1 \xrightarrow{Rw} des_2 \xrightarrow{Rw} des_3 \xrightarrow{Rw} \cdots \xrightarrow{Rw} des_n = des'$$

3. Calculate $k := \pi(v(des'))$ and return k.

2.3 Problem Solving Based on Equivalent Transformation

An **equivalent transformation** (ET) rule is defined as a rewriting rule r that satisfies $v(des_1) = v(des_2)$ for all pairs (des_1, des_2) in r. In order to transform elements in Des equivalently, ET rules are used.

Theorem 1. *If the set Rw consists only of ET rules, the Algorithm A(Rw) gives a correct answer for any problem α.*

A set P of rewriting rules can be regarded as a **program**, since P determines a (possibly nondeterministic) algorithm based on the Algorithm A(Rw). If a program P consists only of ET rules, then computation by P is correct, i.e., a correct answer for Problem α is obtained.

3 Separated Descriptions

A **separated representation system** is a six-tuple

$$\langle Ds, Qs, w, Ms, m, Meg \rangle$$

of two sets Ds and Qs for descriptions, two sets Ms and Meg for meanings, a mapping w from Ds to Ms, and a mapping m from $Ms \times Qs$ to Meg.

Assume that $\langle Ds, Qs, w, Ms, m, Meg \rangle$ is a separated representation system. If we define a set Des as $Ds \times Qs$, and a mapping v from Des to Meg by

$$v(des) = m(w(d), q)$$

for all $des = (d, q)$ in Des, then $\langle Des, v, Meg \rangle$ is obviously a representation system, which is called the **associated representation system** of the separated representation system $\langle Ds, Qs, w, Ms, m, Meg \rangle$. A separated representation system $\langle Ds, Qs, w, Ms, m, Meg \rangle$ is always identified with its associated representation system. Hereafter we assume that we are given a separated representation system $\Gamma = \langle Ds, Qs, w, Ms, m, Meg \rangle$.

A subset r of $Qs \times Qs$ and an element d in Ds determines a rewriting rule r':

$$r' = \{((d, q), (d, q')) \mid (q, q') \in r\},$$

which is called the **associated rewriting rule** of r with respect to d. A subset r of $Qs \times Qs$ is called a **rewriting rule** on Qs. A rewriting rule r on Qs is called an **ET rule** with respect to an element d in Ds iff the associated rewriting rule r' of r with respect to d is an ET rule. Obviously a rewriting rule r on Qs is an **ET rule** with respect to d in Ds iff

$$m(w(d), q) = m(w(d), q')$$

for all $(q, q') \in r$. A description (d, q) in $Ds \times Qs$ is equivalently transformed into (d, q') in $Ds \times Qs$ by an ET rule r on Qs with respect to d in Ds.

4 Program Synthesis by Equivalent Transformation

4.1 Specification and ET-rule-set Generator

A **specification** on Γ is a pair (d, Q) of d in Ds and a subset Q of Qs. A specification (d, Q) on Γ requires a program to answer all problems (d, q) such that $q \in Q$.

A mapping g from Ds to the powerset of the set of all rewriting rules on Qs is called a **rewriting-rule-set generator** on Γ. For all d in Ds, a rewriting-rule-set generator g on Γ determines a set $g(d)$ of rewriting rules on Qs.

A rewriting-rule-set generator g on Γ is called an **ET-rule-set generator** on Γ iff, for all d in Ds, each element in $g(d)$ is an ET rule with respect to d.

4.2 Obtaining ET-Rules by Equivalent Transformation

Theorem 2. *Assume that g is an ET-rule-set generator on Γ. If two elements d and d' in Ds satisfies $w(d) = w(d')$, then $g(d')$ is a set of ET rules with respect to d.*

Since w is a mapping from Ds to Ms, $\langle Ds, w, Ms\rangle$ is a representation system. A relation $\sim$ on Ds is defined by

$$d_1 \sim d_2 \iff w(d_1) = w(d_2).$$

Obviously $\sim$ is an equivalence relation. According to the general definitions of rewriting rules and ET rules, a rewriting rule r on Ds is an ET rule on Ds iff $w(d) = w(d')$ for all (d, d') in r.

Algorithm B(Rw, g)

Let Rw be a set of rewriting rules on Ds, and g a rewriting-rule-set generator on Γ.
Input: a specification (d_0, Q) on Γ
Output: a set of rewriting rules on Qs.

1. Transform d_0 into d_n by repeated application of rewriting rules in Rw, i.e.,

$$d_0 \xrightarrow{Rw} d_1 \xrightarrow{Rw} \cdots \xrightarrow{Rw} d_n.$$

   Rewriting rules may be applied any finite times ($n \geq 0$) as long as they are applicable.
2. From d_n, obtain a set of rewriting rules $g(d_n)$ by using the rewriting-rule-set generator g.

Theorem 3. *If all rewriting rules in Rw are ET rules and g is an ET-rule-set generator on Γ, then the set of rewriting rules obtained by Algorithm B(Rw, g) is a set of ET rules with respect to d_0.*

5 Concluding Remarks

A theoretical basis for program synthesis by equivalent transformation has been developed. Given a specification (d_0, Q) on a separated representation system $\Gamma = \langle Ds, Qs, w, Ms, m, Meg\rangle$, a program is obtained by transforming d_0 equivalently into d_n and by mapping d_n using an ET-rule-set generator g. An element d in Ds is transformed in **program synthesis** preserving meaning $w(d)$ of d, while an element q in Q is transformed in **computation** preserving meaning $m(w(d), q)$ of (d, q). This theory can be applied to many declarative programs including logic and functional programs.

References

1. Akama,K., Shigeta,Y. and Miyamoto,E. : A Framework of Problem Solving by Equivalent Transformation of Logic Program, J. Japan Soc. Artif. Intell., Vol.12, No.2, pp.90–99 (1997).
2. Futamura, Y.: Partial Evaluation of Computation Process — an Approach to a Compiler-compiler. Systems. Computers. Controls. Vol.25. (1971) 45–50
3. J.W. Lloyd, Foundations of Logic Programming, Second edition, Springer-Verlag, 1987.
4. A. Pettorossi and M. Proietti, "Transformation of Logic Programs: Foundations and Techniques", *The Journal of Logic Programming*, Vol.19/20, 1994, pp.261–320.

Equivalent Transformation by Safe Extension of Data Structures

Kiyoshi Akama[1] and Hidekatsu Koike[2] and Hiroshi Mabuchi[3]

[1] Hokkaido University, Kita 11, Nishi 5, Kita-ku, Sapporo, 060-0811, Japan,
e-mail: akama@cims.hokudai.ac.jp,

[2] Hokkaido University, Kita 11, Nishi 5, Kita-ku, Sapporo, 060-0811, Japan,
e-mail: koke@cims.hokudai.ac.jp,

[3] Iwate Prefectural University, 152-52 Sugo Takizawa Iwate 020-0173 Japan,
e-mail: mabu@soft.iwate-pu.ac.jp

Abstract. Equivalent transformation is one of the most important methods for improving efficiency of programs. However, no general theoretical foundation has been developed for improving efficiency of programs by changing data structures in the programs. In this paper we develop a theoretical foundation of equivalent transformation that introduces new data structures. We define a concept of safe extension of data structures, and prove that the meaning of a description (a declarative program) on a data structure is preserved by safe extension of the data structure.

1 Introduction

It is well known that efficiency of computation depends on data structures. In case of procedural programming languages, it is taken for granted to adopt better data structures for efficient computation [6]. Data structures are also important in logic paradigm. For instance, expressive power and efficiency is improved by using class variables based on sort hierarchy [1, 2]. Moreover, most of constraint satisfaction problems cannot be solved within reasonable time in Prolog, while constraint logic programs with domain variables solve them far more efficiently [3]. There are many programs that can be improved in efficiency by introducing new data structures.

Equivalent transformation is one of the most important methods for improving efficiency of programs [5]. Declarative programs such as logic and functional programs can be improved to be more efficient ones by using unfolding, folding, goal replacement, tupling, and other transformations. In most transformations programs are changed, while data structures used in the programs are usually not changed. Very few theoretical foundations have been developed for improving efficiency of programs by changing data structures in the programs.

In this paper we develop a theoretical foundation of equivalent transformation by introducing new data structures, based on which we can often make programs more efficient. For instance, in the class-variable example [1, 2], improvement of efficiency is obtained by the following equivalent transformations:

1. (ET_1) introduction of class variables,
2. (ET_2) transformation of programs using class variables.

The domain-variable example [3] is similarly improved by the following two steps:

1. (ET_1) introduction of domain variables,
2. (ET_2) transformation of programs using domain variables.

In both cases, introduction of new data structures (ET_1) is essential to further transformation (ET_2).

The purpose of the paper is to formalize the first equivalent transformation (ET_1). We define a structure called a **specialization system**, which is a base structure for specifying data structures in problem description. We also introduce **declarative descriptions** on a specialization system. A declarative description d on a specialization system Γ is associated with its meaning $\mathcal{M}(\Gamma, d)$. Equivalent transformation is a transformation of (Γ, d) preserving $\mathcal{M}(\Gamma, d)$. We have two kinds of equivalent transformation for a pair (Γ, d) of a declarative description d and a specialization system Γ:

1. ET_1: $(\Gamma_1, d) \rightarrow (\Gamma_2, d)$ change from Γ_1 into Γ_2 with d unchanged,
2. ET_2: $(\Gamma, d_1) \rightarrow (\Gamma, d_2)$ change from d_1 into d_2 with Γ unchanged.

We define **extension** and **safe extension** of specialization systems to formulate introduction of new data structures. We also prove the correctness of equivalent transformation by safe extension of specialization systems.

2 Declarative Description

2.1 Definition of Specialization Systems

Definition 1. *A specialization system is a four-tuple $\langle \mathcal{A}, \mathcal{G}, \mathcal{S}, \mu \rangle$ of three sets $\mathcal{A}$, $\mathcal{G}$ and $\mathcal{S}$, and a mapping $\mu : \mathcal{S} \to partial\_map(\mathcal{A})$ that satisfies the following requirements, where $partial\_map(X)$ is the set of all partial mappings on X.*

1. $\forall s_1, s_2 \in \mathcal{S}, \exists s \in \mathcal{S} : \mu(s) = \mu(s_2) \circ \mu(s_1)$.
2. $\exists s \in \mathcal{S}, \forall a \in \mathcal{A} : \mu(s)(a) = a$.
3. $\mathcal{G} \subseteq \mathcal{A}$.

Elements of $\mathcal{A}$ are called atoms. Elements of $\mathcal{G}$ are called ground atoms. Elements of $\mathcal{S}$ are called specializations.

When there is no danger of confusion, elements in $\mathcal{S}$ are regarded as partial mappings over $\mathcal{A}$ and the following notational convention is used. Each element in $\mathcal{S}$ is identified as a partial mapping on $\mathcal{A}$, and the application of such a partial mapping is represented by postfix notation. For example, $\theta \in \mathcal{S}$ and $\mu(\theta)(a)$ are denoted respectively by $\theta \in \mathcal{S}$ and $a\theta$.

2.2 Declarative Description

Definition 2. *Let Γ be a specialization system $\langle \mathcal{A}, \mathcal{G}, \mathcal{S}, \mu \rangle$. A definite clause on Γ is a formula of the form:*

$$H \leftarrow B_1, B_2, \cdots, B_n$$

where $H, B_1, B_2, \cdots, B_n$ are atoms in $\mathcal{A}$. A declarative description on Γ is a set of definite clauses on Γ.

Let C be a definite clause on X. The head of a clause C is denoted by $head(C)$, and the set of all atoms in the body of C is denoted by $body(C)$.

A specialization $s \in \mathcal{S}$ is applicable to $a \in \mathcal{A}$ iff there exists $b \in \mathcal{A}$ such that $\mu(s)(a) = b$. When $\theta \in \mathcal{S}$ is applicable to $H, B_1, B_2, \cdots, B_n$, a definite clause $C\theta = (H\theta \leftarrow B_1\theta, B_2\theta, \cdots, B_n\theta)$ is obtained from a definite clause $C = (H \leftarrow B_1, B_2, \cdots, B_n)$. A definite clause C' is an instance of C iff there is a specialization θ such that $C' = C\theta$. A definite clause C is ground iff it consists of only ground atoms. A ground instance of a definite clause C is a ground definite clause that is an instance of C. Let P be a declarative description on Γ. The set of all ground instances of definite clauses in P is denoted by $Gclause(P)$.

2.3 Meaning of a Declarative Description

For a declarative description P, the meaning $\mathcal{M}(P)$ is defined by

$$\mathcal{M}(P) = \bigcup_{n=0}^{\infty} [T_P]^n(\emptyset),$$

where T_P is a mapping on the powerset $2^{\mathcal{G}}$, which mapps a subset of $\mathcal{G}$ into another subset of $\mathcal{G}$:

$$T_P(x) = \{head(C) \mid body(C) \subseteq x, C \in Gclause(P)\}.$$

Since $Gclause(P)$ and T_P depend on the specialization system Γ, $\mathcal{M}(P)$ also depends on Γ. Hereafter when Γ should be specified explicitly, $\mathcal{M}(P)$ will be denoted by $\mathcal{M}(\Gamma, P)$.

3 Preservation of Meaning by Safe Extension

We consider two specialization systems Γ_1 and Γ_2, and discuss the relationship between the two meanings of a declarative description on the two specialization systems.

3.1 Extension of Specialization Systems

Consider a mapping $\mu : \mathcal{S} \to partial\_map(\mathcal{A}_d)$. The mapping μ determines a subset μ' of $\mathcal{S} \times \mathcal{A} \times \mathcal{A}$ uniquely by

$$\mu' = \{(s, a, b) \mid s \in \mathcal{S}, a \in \mathcal{A}, b \in \mathcal{A}, \mu(s)(a) = b\}.$$

It is obvious that the mapping that determines μ' from μ is one-to-one. In this paper μ will be identified with μ', thus μ will be regarded as a subset of $\mathcal{S} \times \mathcal{A} \times \mathcal{A}$, which is convenient for discussion of the relation between two specialization systems.

Definition 3. *Let Γ_1 and Γ_2 be specialization systems:*

$\Gamma_1 = \langle \mathcal{A}_1, \mathcal{G}_1, \mathcal{S}_1, \mu_1 \rangle,$
$\Gamma_2 = \langle \mathcal{A}_2, \mathcal{G}_2, \mathcal{S}_2, \mu_2 \rangle.$

Γ_2 is an **extension** *of Γ_1 (or Γ_1 is a* **partial specialization system** *of Γ_2), iff $\mathcal{A}_1 \subseteq \mathcal{A}_2$, $\mathcal{G}_1 \subseteq \mathcal{G}_2$, $\mathcal{S}_1 \subseteq \mathcal{S}_2$, and $\mu_1 \subseteq \mu_2$.*

Note that μ_1 and μ_2 are regarded, respectively, as subsets of $\mathcal{S}_1 \times \mathcal{A}_1 \times \mathcal{A}_1$ and $\mathcal{S}_2 \times \mathcal{A}_2 \times \mathcal{A}_2$, and that $\mu_1 \subseteq \mu_2$ iff any elements (θ, a, b) in μ_1 are included also in μ_2.

3.2 Inclusion Relation of Meaning

The following theorem states that the meaning of a declarative description on a specialization system increases by extension of the specialization system.

Theorem 1. *Assume that Γ_2 is an extension of Γ_1. If P is a declarative description on Γ_1, then P is also a declarative description on Γ_2, and*

$$\mathcal{M}(\Gamma_1, P) \subseteq \mathcal{M}(\Gamma_2, P).$$

3.3 Safe Extension

Definition 4. *Let Γ_1 and Γ_2 be specialization systems:*

$\Gamma_1 = \langle \mathcal{A}_1, \mathcal{G}_1, \mathcal{S}_1, \mu_1 \rangle,$
$\Gamma_2 = \langle \mathcal{A}_2, \mathcal{G}_2, \mathcal{S}_2, \mu_2 \rangle.$

Γ_2 is a **safe extension** *of Γ_1 iff the following conditions are satisfied.*

1. *Γ_2 is an extension of Γ_1.*
2. *$\mathcal{G}_1 = \mathcal{G}_2 = \mathcal{G}$.*
3. *For any finite subset X of $\mathcal{A}_1$ and for any specialization θ in $\mathcal{S}_2$ such that $X\theta \subseteq \mathcal{G}$, there is a specialization σ in $\mathcal{S}_1$ such that $x\theta = x\sigma$ for any $x \in X$.*

3.4 Preservation of Meaning

The following theorem states that the meaning of a declarative description on a specialization system is preserved by safe extension of the specialization system.

Theorem 2. [Safe-extension Theorem] *Assume that Γ_2 is a safe extension of Γ_1. If P is a declarative description on Γ_1, then P is also a declarative description on Γ_2 and*
$\mathcal{M}(\Gamma_1, P) = \mathcal{M}(\Gamma_2, P)$.

4 Conclusion

In this paper, we first define specialization systems and declarative descriptions. A problem is formalized as a pair of a specialization system and a declarative description. If we change a specialization system Γ_1 into Γ_2 with a declarative description d left unchanged, (Γ_1, d) is transformed into (Γ_2, d). If Γ_2 is a safe extension of Γ_1, the transformation from (Γ_1, d) to (Γ_2, d) is an equivalent transformation, i.e., $\mathcal{M}(\Gamma_1, d) = \mathcal{M}(\Gamma_2, d)$. This theory can be applied to many examples of efficiency improvement by introducing new data structures including class-variable examples and domain-variable examples.

References

1. Aït-Kaci, H. and Nasr, R. : *LOGIN: A Logic Programming Language with Built-In Inheritance*, The Journal of Logic Programming, 3 (1986).
2. Akama,K. : *PAL: An Extended Prolog with Inheritance Hierarchy*, information processing society of Japan, Vol.28 No.4 pp.27-34 (1987).
3. Hentenryck,V. : *Constraint Satisfaction in Logic Programming*, The MIT Press (1989).
4. Lloyd, J.W.: *Foundations of Logic Programming*, Second edition, Springer-Verlag (1987).
5. Pettorossi,A. and Proietti,M., "Transformation of Logic Programs: Foundations and Techniques", *The Journal of Logic Programming*, Vol.19/20, 1994, pp.261–320.
6. Wirth, N. : *Algorithms + Data Structures = Programs*, Prentice-Hall (1976).

Semantics and Transformations in Formal Synthesis at System Level[*]

Viktor Sabelfeld, Christian Blumenröhr, Kai Kapp

Institute of Computer Design and Fault Tolerance, Karlsruhe University
e-mail: {sabelfel,blumen,kai.kapp}@ira.uka.de Web: http://goethe.ira.uka.de/fsynth

Abstract. In formal synthesis methodology, circuit implementations are derived from specifications by means of elementary logical transformation steps, which are performed within a theorem prover. In this approach, additionally to the circuit implementation, the proof that the result is a correct implementation of a given specification is obtained automatically. In this paper, we formally describe the functional semantics of system specifications in higher order logic. This semantics build the basis for formal synthesis at system level. Further, theorems for circuit optimisation at this level are proposed.

1 Introduction

The most critical question in circuit synthesis is the correctness of the design: how can one guarantee the correctness of the automatically generated circuit implementation with regard to a given specification? The synthesis programs are too big and too complex to prove their correctness using the available software verification tools.

In formal synthesis, the circuit implementation is obtained from the specification by the application of elementary transformation rules which are formulated in higher order logic and proved as theorems in a theorem prover. The correctness of a transformation means a mathematical relation between the source and the result. If such correct transformations are used during the synthesis process, then, as a result, not only the circuit implementation, but also a proof of the correctness of this implementation ("correctness by construction") is obtained.

In the formal synthesis, most approaches deal with the synthesis of digital systems at lower levels of abstraction [9], or with synthesis of pure data-flow descriptions at the algorithmic level [7].

Besides formal synthesis, there are approaches which can be summarized by the term "transformational design". They also claim to fulfill the paradigm of "correctness by construction": The synthesis process is also based on correctness-preserving transformations. However, the transformations are proved by paper & pencil and afterwards implemented in a complex software program: It is implicitly assumed that an automatic synthesis process is correct by definition. But there is no guarantee that the transformations have been implemented correctly and therefore, the correctness of the synthesis process cannot be regarded as proved. A successfully applied approach for synthesis at the system level that falls into this category is described in [5].

In our approach to formal synthesis, we have developed the language *Gropius* [1] to describe the circuit specifications and implementations .

The BNF below describes the syntactic structure of DFG-terms (acyclic **D**ata **F**low **G**raphs). They represent non-recursive programs that always terminate.

$$\begin{aligned}
\mathit{var\_struct} &::= \mathit{variable}\ \left[\text{“:”}\mathit{type}\right] \mid \text{“(”}\ \mathit{var\_struct}\ \text{“,”}\ \mathit{var\_struct}\ \text{“)”}\\
\mathit{expr} &::= \mathit{var\_struct} \mid \mathit{constant}\ \left[\text{“:”}\mathit{type}\right] \mid \text{“(”}\ \mathit{expr}\ \text{“,”}\ \mathit{expr}\ \text{“)”} \mid \text{“(”}\ \mathit{expr}\ \mathit{expr}\ \text{“)”}\\
\mathit{DFG\text{-}term} &::= \text{“}\lambda\text{”}\ \mathit{var\_struct}\ \text{“.”}\ \left[\ \text{“let”}\ \mathit{var\_struct} = \mathit{expr}\ \text{“in”}\ \right]\ \mathit{expr}
\end{aligned}$$

A *condition* is a DFG-term which produces a boolean output. We fixed the following syntax for program terms in Gropius:

$$\begin{aligned}
\mathit{P\text{-}term} ::= {} & \text{“PARTIALIZE”}\ \mathit{DFG\text{-}term} \mid \text{“WHILE”}\ \mathit{condition}\ \mathit{P\text{-}term}\\
& \mid \mathit{P\text{-}term}\ \text{“SERIAL”}\ \mathit{P\text{-}term} \mid \mathit{P\text{-}term}\ \text{“PARALLEL”}\ \mathit{P\text{-}term}\\
& \mid \text{“IF”}\ \mathit{condition}\ \mathit{P\text{-}term}\ \mathit{P\text{-}term} \mid \text{“LEFT”}\ \mathit{P\text{-}term} \mid \text{“RIGHT”}\ \mathit{P\text{-}term}
\end{aligned}$$

The P-terms have a type $\alpha \to \beta\,\mathsf{partial}$. To represent the data type of a P-term which may not terminate, the type $\alpha\,\mathsf{partial}$ has been introduced. It extends an arbitrary type α by a new value $\bot$: $\mathsf{partial} = \bot \mid \mathsf{Def\ of}\ \alpha$. The function Case: $(\mathsf{Case}\,\bot\,f\,a \overset{\text{def}}{=} a) \wedge (\mathsf{Case}\,(\mathsf{Def}\,x)\,f\,a \overset{\text{def}}{=} f\,x)$ is used to define functions on values of type $\alpha\,\mathsf{partial}$. More details about P-terms and the algorithmic level of Gropius can be found in [2].

[*] This work has been partly financed by the Deutsche Forschungsgemeinschaft, Project SCHM 623/6-3.

An algorithmic description expresses the functional dependencies of outputs on the inputs of the circuit. It does not take time into account. During high-level synthesis the algorithmic description is mapped to an register transfer (RT) level structure. To bridge the gap between these two different abstraction levels one has to determine how the circuit communicates with its environment. Therefore, as a second component of the circuit representation an interface description is required. The interface description defines the temporal signal behavior at the circuit interface and specifies exactly the communication of the circuit with the environment. More details about interface patterns can be found in [2, 3].

2 Circuit Descriptions at the System Level

Below we give the definitions for the syntax and semantics of system descriptions called here System-structures or S-structures for short. In our approach to synthesis at system level, all the processes interact via a fixed communication scheme which is label-based and inspired by higher order Petri nets [6]. Our process corresponds to a Petri net transition with some places as inputs and outputs. "Firing" means here removing the input labels, performing some calculation and delivering the result as a new label to the output places.

At the system level, processes communicate via channels. Each channel consists of three signals: a signal $data : \alpha$, and two control signals $data\_valid, ready$: bool. A channel has a fixed direction, defined by the direction of the signal *data*. The *data_valid*-signal goes to the same direction whereas the *ready*-signal goes to the opposite direction.

In channels, communication is performed via handshake. Let us consider a channel from a process A to a process B. A signals via *data_valid* that there is a label with some data being on *data*. Process B signals via *ready* that it is ready to read the next label. Whenever both *data_valid* and *ready* become true, the communication takes place, i.e. the label is moved from A to B.

Four kinds of processes are used as components in S-structures: DFG-term and P-term based processes, K-processes (see Section 3) and S-calls. The DFG- and P-term based processes are essentially nothing else but circuit descriptions at the algorithmic level. The only difference is that the functional description is combined with a special interface pattern for the system level. Both DFG-term based processes and P-term based processes have a single input channel and a single output channel. K-processes are a sort of a glue logic for building arbitrary S-structures and for managing the communication between other processes. They may have an arbitrary (but fixed) number of input and output signals and are used to spread, combine, buffer, delay or synchronize signals. Finally, S-calls are nothing else but procedure calls. One can declare (non-recursive) S-definitions and give them arbitrary names. S-definitions allow the use of hierarchical circuit descriptions in Gropius and reduce the complexity of the synthesis process. The syntax of S-structures is as follows:

$$\begin{aligned}
\textit{interface} &::= \text{“(”}\ \textit{channel}\ \{\ \text{“,”}\ \textit{channel}\ \}\ \text{“)”} \\
\textit{DFG-process} &::= \text{“Dfg\_proc”}\ \textit{DFG-term interface} \\
\textit{P-process} &::= \text{“P\_proc”}\ \textit{P-term interface} \\
\textit{K-process} &::= \textit{K-process-name}\ [\ \textit{constant}\ ]\ \textit{interface} \\
\textit{S-call} &::= \textit{structure-name interface} \\
\textit{S-process} &::= \textit{DFG-process}\ \mid\ \textit{P-process}\ \mid\ \textit{K-process}\ \mid\ \textit{S-call} \\
\textit{S-structure} &::= [\ \text{“}\exists\text{”}\ \{\ \textit{channel}\ \}\ \text{“.”}]\ \textit{S-process}\ \{\ \text{“}\wedge\text{”}\ \textit{S-process}\ \} \\
\textit{S-definition} &::= \textit{structure-name interface}\ \text{“=”}\ \textit{S-structure}
\end{aligned}$$

3 K-Processes

We have defined eight elementary K-processes in Gropius: Double, Join, Synchronize, Split, Fork, Choose, Source, and Sink. In K-processes, no calculations on data labels are performed. Labels are solely moved according to the label based communication pattern presented in the previous section.

The K-process Double duplicates the input label, Join combines two separate labels into a single paired label, Split is the inverse process to Join.

The K-process Synchronize collects two labels. As soon as both successor processes are ready, both labels are given over simultaneously.

The K-process Fork delivers the first component of an incoming label of type $\alpha \times$ bool to one of the output channels, depending on whether the second component of the label is false or true.

The K-process Choose has two input channels $in_1, in_2 : \alpha \times$ bool and one output channel $out : \alpha$. It is the only process, which can fire even if not all inputs are ready. Choose delivers to its successor the first value of a label it became either from in_1 or in_2. There are two different states, Ready_1 and Ready_2, in which Choose can fire. In the state Ready_i, only channel in_i is ready. Every time a data label (d, b) occurs, firing delivers the label d to the output channel *out*; it leaves the state unchanged, if $b = \mathsf{T}$, or changes it to Ready_{3-i}, if $b = \mathsf{F}$.

The process Source is a source in the data flow and yields a constant every time the output channel is ready to receive a signal. The process Sink has one input channel but no output channels and implements a sink of a signal.

4 Functional Semantics of S-structures

In [3], we have defined the behavioral semantics and equivalence relation for S-structures, which consider the temporal input-output signal behavior. But, an exact definition of the signal flow in time, as given in the behavioral semantics, is often not desirable. To take the purely functional aspects of the system descriptions into account, we will define the functional semantics and the functional equivalence relation for S-structures. Informally, the functional semantics fixes only the sequences of an S-structure's output signals for given sequences of input signals. We represent these (finite or infinite) signal sequences in the theorem prover HOL as values of the type abstraction $\alpha\,\mathsf{signal}$, which are functions f $:\mathsf{num} \to \alpha\,\mathsf{partial}$ satisfying the following *signal condition*: $\mathsf{Is\_signal}\,f \overset{\mathrm{def}}{=} \forall n.\,(f\,n = \bot \Rightarrow \mathsf{Idle}\,f\,n)$, where $\mathsf{Idle}\,f\,n \overset{\mathrm{def}}{=} \forall k.\,(n \leq k \Rightarrow f\,k = \bot)$. Here, the value $\bot$ stands for the absence of any signal value. Using the type definition package by Melham [8] we have defined a representation function $\mathsf{from\_signal} : \alpha\,\mathsf{signal} \to (\mathsf{num} \to \alpha)$ and its inverse abstraction function $\mathsf{to\_signal}$, so that the following theorem holds:

$$\vdash (\forall x : \alpha\,\mathsf{signal}.\ \mathsf{to\_signal}\,(\mathsf{from\_signal}\,x) = x) \wedge$$
$$(\forall f : \mathsf{num} \to \alpha\,\mathsf{partial}.\ \mathsf{Is\_signal}\,f = (\mathsf{from\_signal}(\mathsf{to\_signal}\,f) = f))$$

In order to build new signals from existing ones we introduce the operator Δ:

$$\Delta\ (x : \alpha\ \mathsf{signal})\ (A : \alpha \to \beta)\ 0 \equiv \mathsf{Case}\ (\mathsf{from\_signal}\ x\ 0)\ A\ \bot$$
$$\Delta\ x\ A\ (\mathsf{SUC}\ n) \equiv \mathsf{Case}\ (\Delta\ x\ A\ n)(\lambda z.\ \mathsf{Case}\ (\mathsf{from\_signal}\ x\ (\mathsf{SUC}\ n))\ A\ \bot)\bot$$

The operator Δ yields functions of natural arguments satisfying the signal condition: $\vdash \forall x\ A.\ \mathsf{Is\_signal}(\Delta\ x\ A)$. So, a signal transformer APPLY can be defined as follows: $\mathsf{APPLY}\ A\ x \overset{\mathrm{def}}{=} \mathsf{to\_signal}(\Delta\ x\ A)$.

The functional equivalence of S-structures introduced below does not take into account the time when signal values are emitted or received. It ignores the names and the values of the intermediate signals as well. The functional semantics $[\![P]\!]$ of an S-structure P is a boolean higher order function. Its input and output parameters are (abstractions of) signal values of the type α signal.

$$\|\ \mathsf{P\_proc}(P : \alpha \to \beta\,\mathsf{partial})\ \|\ (x : \alpha\,\mathsf{signal}, y : \beta\,\mathsf{signal}) \overset{\mathrm{def}}{=} (y = \mathsf{APPLY}\ P\ x)$$
$$\|\ \mathsf{Dfg\_proc}\,f\ \|\ (x : \alpha\,\mathsf{signal}, y : \beta\,\mathsf{signal}) \overset{\mathrm{def}}{=} (y = \mathsf{APPLY}\,(\mathsf{Def} \circ f)\,x)$$
$$\|\ \mathsf{Double}\ \|\ (x : \alpha\,\mathsf{signal}, y_1 : \alpha\,\mathsf{signal}, y_2 : \alpha\,\mathsf{signal}) \overset{\mathrm{def}}{=} (y_1 = x) \wedge (y_2 = x)$$
$$\|\ \mathsf{Split}\ \|\ (x : (\alpha \times \beta)\,\mathsf{signal}, y_1, y_2) \overset{\mathrm{def}}{=}$$
$$(y_1 = \mathsf{APPLY}\,(\mathsf{Def} \circ \mathsf{FST})\,x) \wedge (y_2 = \mathsf{APPLY}\,(\mathsf{Def} \circ \mathsf{SND})\,x)$$

The definitions for the functional semantics of K-processes Join, Synchronize, Fork, Choose, Source and Sink can be found in Appendix A.

The functional semantics of an S-structure $\mathcal{S} = \exists\ \bar{b}.\ S_1 \wedge \ldots \wedge S_n$ is defined by $\|\ \mathcal{S}\ \|\ \overset{\mathrm{def}}{=} \exists\ \bar{b}.\ \ \|\ S_1\ \|$ $\wedge \ldots \wedge \|\ S_n\ \|$. We call two S-structures S_1 and S_2 *functional equivalent*, if their functional semantics are equal: $S_1 \approx S_2 \overset{\mathrm{def}}{=}\ \|\ S_1\ \| = \|\ S_2\ \|$.

5 Program Transformations

In what follows we present some functional equivalence theorems which can be considered as Gropius program transformations and build the basis for circuit combining/partitioning algorithms. All of the theorems have been proved in the theorem prover HOL [4].

$$\vdash \exists r\ s.\ \mathsf{Double}\,(x : \alpha\,\mathsf{signal}, r, s) \wedge \mathsf{P\_proc}\ A\,(r, u) \wedge \mathsf{P\_proc}\ A\,(s, v)$$
$$\approx \tag{1}$$
$$\exists\,(t : \beta\,\mathsf{signal}).\ \mathsf{P\_proc}\,(A : \alpha \to \beta\,\mathsf{partial})(x, t) \wedge \mathsf{Double}\,(t, u, v)$$

By the application of theorem (1) the double execution of a P-process is avoided: two copies are combined into a single one. Theorem (2) allows to combine two P-processes, which are executed in sequence, into a single P-process, or to partition a given P-process into two successively executed P-processes.

$$\vdash (\exists u.\, \mathsf{P\_proc}\ A\ (x,u) \land \mathsf{P\_proc}\ B\ (u,y)) \approx (\mathsf{P\_proc}\ (A\ \mathsf{SERIAL}\ B)\ (x,y)) \tag{2}$$

Theorem (3) describes the functional equivalence of an S-structure consisting of two parallel P-processes, whose results are combined over a K-process Join, to a structure where first the two inputs are combined using Join into a single value which is then forwarded to a single P-process. With the help of these theorems a P-process can be partitioned into two parallel ones or two different P-processes can be combined to a single one.

$$\begin{array}{c}\vdash (\exists u\ v.\, \mathsf{P\_proc}\, A\ (x,u) \land \mathsf{P\_proc}\, B\ (y,v) \land \mathsf{Join}(u,v,z)) \\ \approx \\ (\exists w.\, \mathsf{Join}(x,y,w) \land \mathsf{P\_proc}\ (A\ \mathsf{PARALLEL}\ B)(w,z))\end{array} \tag{3}$$

6 Conclusion

We have presented a method for the formal synthesis at the system level. In our approach, systems are described as structures of concurrent processes. The synthesis of correct circuits is performed by means of equivalent transformations, which are applications of theorems in the theorem prover HOL. For the proof of the correctness of transformations, the functional equivalence relation on process structures has been introduced.

References

[1] C. Blumenröhr and D. Eisenbiegler. *Performing High-Level Synthesis via Program Transformations within a Theorem Prover*, In: Digital System Design Workshop at the 24th EUROMICRO 98 Conference, 1998.

[2] C. Blumenröhr and V. Sabelfeld. *Formal Synthesis at the Algorithmic Level*, In: Correct Hardware Design and Verification Methods, Charme'99, 1999.

[3] C. Blumenröhr. *A formal approach to specify and synthesize at the system level*, In: M. Mutz and N. Lange, eds., GI/ITG/GME Workshop Modellierung und Verifikation von Schaltungen und Systemen, Braunschweig, Germany, February 1999, pp. 11-20, Shaker-Verlag.

[4] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, Cambridge University Press, 1993.

[5] J. Iyoda, A. Sampaio, L. Silva. *ParTS: A Partitioning Transformation System*, In: FM'99 — Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, LNCS 1709, pp. 1400–1419.

[6] K. Jensen. *Colored Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, Vol. 1: Basic Concepts, Springer Verlag 1992.

[7] M. Larsson. *An engineering approach to formal digital system design.* The Computer Journal, 38(2):101–110, 1995.

[8] T.F. Melham. *Automating Recursive Type Definitions in Higher Order Logic*, In: G. Birtwistle and P. A. Subrahmanyam, eds., Current Trends in Hardware Verification and Automated Theorem Proving, pp. 341-386, Springer-Verlag, 1989.

[9] R. Sharp, O. Rasmussen. *The T-Ruby design system.* In: IFIP Conference on Hardware Description Languages and their Applications, pp. 587–596, 1995.

A Functional Semantics of K-processes Join, Synchronize, Fork, Choose, Source and Sink

Similar to Δ and APPLY, the operator Δ^2 for building new paired value signals and the signal transformer $\mathsf{APPLY}^2\ B\ x\ y \overset{\mathrm{def}}{=} \mathsf{to\_signal}(\Delta^2\ x\ y\ B)$ can be introduced in such a way that $\forall x\, y\, B.\ \mathsf{Is\_signal}(\Delta^2\ x\, y\, B)$ holds.

$$\begin{array}{l}\mathsf{Case}^2\ (a : \alpha\ \mathsf{partial})(b : \beta\ \mathsf{partial})(B : \alpha \times \beta \to \gamma\ \mathsf{partial}) \overset{\mathrm{def}}{=} \\ \qquad \mathsf{Case}\ a\ (\lambda u.\ \mathsf{Case}\ b\ (\lambda v. B(u,v))\ \bot)\ \bot \\ \Delta^2\ (x : \alpha\ \mathsf{signal})(y : \beta\ \mathsf{signal})(B : \alpha \times \beta \to \gamma\ \mathsf{partial})\ 0 \overset{\mathrm{def}}{=} \\ \qquad \mathsf{Case}^2\ (\mathsf{from\_signal}\ x\ 0)(\mathsf{from\_signal}\ y\ 0)\ B \\ \Delta^2\ x\ y\ B\ (\mathsf{SUC}\ n) \overset{\mathrm{def}}{=} \\ \quad (\mathsf{Case}\ (\Delta^2\ x\ y\ B\ n)\ (\mathsf{Case}^2\ (\mathsf{from\_signal}\ x\ (\mathsf{SUC}\ n))(\mathsf{from\_signal}\ y\ (\mathsf{SUC}\ n))\ B)\ \bot\end{array}$$

Now the definitions for the functional semantics of Join and Synchronize follow.

$$\begin{aligned}
&\| \,\mathsf{Join}\, \| \,(x_1 : \alpha\,\mathsf{signal}, x_2 : \beta\,\mathsf{signal}, y) \stackrel{\mathrm{def}}{=} (y = \mathsf{APPLY}^2\ \mathsf{Def}\ x_1\ x_2)\\
&\| \,\mathsf{Synchronize}\, \| \,(x_1 : \alpha\,\mathsf{signal}, x_2 : \beta\,\mathsf{signal}, y_1 : \alpha\,\mathsf{signal}, y_2 : \beta\,\mathsf{signal}) \stackrel{\mathrm{def}}{=}\\
&\quad (y_1 = \mathsf{APPLY}^2\ (\mathsf{Def} \circ \mathsf{FST})\ x_1\ x_2) \wedge (y_2 = \mathsf{APPLY}^2\ (\mathsf{Def} \circ \mathsf{SND})\ x_1\ x_2)
\end{aligned}$$

To define the functional semantics of the K-process Fork we have introduced an auxiliary function $LenF$. $LenF(x : (\alpha \times \mathsf{bool})\,\mathsf{signal})\,n$ delivers a pair (l_1, l_2) of natural numbers where $l_1 (l_2)$ is the number of truth values F (resp.T) in the sequence $\mathsf{SND}(\mathsf{from\_signal}\ x\ 0), \ldots, \mathsf{SND}(\mathsf{from\_signal}\ x\ n)$.

$$\begin{aligned}
&\| \,\mathsf{Fork}\, \| \,(x : (\alpha \times \mathsf{bool})\,\mathsf{signal}, y_F : \alpha\,\mathsf{signal}, y_T : \alpha\,\mathsf{signal}) \stackrel{\mathrm{def}}{=} \mathsf{let}\ \emptyset = (\lambda n.\ 0)\\
&\mathsf{and}\ \perp\!\!\perp\ = \mathsf{from\_signal}\,(\lambda n.\ \perp)\ \mathsf{in}\ (\forall n.\ \mathsf{let}\ (l_F, l_T) = LenF\,x\,n\ \mathsf{in}\ \mathsf{Case}\ (\mathsf{from\_signal}\,x\,n)\\
&(\mathsf{Def} \circ \mathsf{FST} = (\lambda z.\ \mathsf{if}\ \mathsf{SND}\ z\ \mathsf{then}\ \mathsf{from\_signal}\ y_T(l_T - 1)\ \mathsf{else}\ \mathsf{from\_signal}\ y_F(l_F - 1)))\\
&(\mathsf{Idle}\ (\mathsf{from\_signal}\ y_F)\ l_F) \wedge (\mathsf{Idle}\ (\mathsf{from\_signal}\ y_T)\ l_T)) \wedge\\
&((\mathsf{FST} \circ (LenF\ x) = \emptyset) \Rightarrow (y_F = \perp\!\!\perp)) \wedge ((\mathsf{SND} \circ (LenF\ x) = \emptyset) \Rightarrow (y_T = \perp\!\!\perp))
\end{aligned}$$

To define the functional semantics of the K-process Choose we have introduced two auxiliary functions $LenC$ and $Choice$. A call $LenC\,n\,x\,y$ delivers a triple $(nextready, l_1, l_2)$, where $nextready = \mathsf{T}$ iff the first input channel is ready after $n+1$ firings of Choose on input channels x and y, and l_1 and l_2 are the numbers of values read from x and y, resp., after these $n+1$ firings.

$$\begin{aligned}
&Choice\ 0\ (x : (\alpha \times \mathsf{bool})\,\mathsf{signal})\ (y : (\alpha \times \mathsf{bool})\,\mathsf{signal})\ (out : \alpha\,\mathsf{signal}) \stackrel{\mathrm{def}}{=}\\
&\quad \mathsf{Case}\ (\mathsf{from\_signal}\ out\ 0)(\lambda z.\ \mathsf{Case}\ (\mathsf{from\_signal}\ x\ 0)(\lambda p.\ z = \mathsf{FST}\ p)\ \mathsf{F})(x = \perp\!\!\perp)\\
&Choice\ (\mathsf{SUC}\ n)\ x\ y\ out \stackrel{\mathrm{def}}{=} \mathsf{let}\ (nextready, l_1, l_2) = LenC\ n\ x\ y\ \mathsf{in}\\
&\quad \mathsf{let}\ s = \mathsf{if}\ nextready\ \mathsf{then}\ \mathsf{from\_signal}\ x\ l_1\ \mathsf{else}\ \mathsf{from\_signal}\ y\ l_2\ \mathsf{in}\\
&\quad \mathsf{Case}\ (\mathsf{from\_signal}\ out\ (\mathsf{SUC}\ n))\ (\lambda z.\ \mathsf{Case}\ s\ (\lambda p.\ (z = \mathsf{FST}\ p))\ \mathsf{F})\\
&\qquad (\mathsf{Idle}\ (\mathsf{from\_signal}\ x)\ l_1) \wedge (\mathsf{Idle}\ (\mathsf{from\_signal}\ y)\ l_2)\\
&\| \,\mathsf{Choose}\, \| \,(x : (\alpha \times \mathsf{bool})\,\mathsf{signal}, y : (\alpha \times \mathsf{bool})\,\mathsf{signal}, out : \alpha\ \mathsf{signal}) \stackrel{\mathrm{def}}{=}\\
&\quad (\forall n.\ Choice\ n\ x\ y\ out) \wedge ((\forall n.\ \mathsf{FST}(LenC\ n\ x\ y)) \Rightarrow (y = \perp\!\!\perp))\\
&\| \,\mathsf{Source}\ (C : \alpha)\, \| \,(out : \alpha\ \mathsf{signal}) \stackrel{\mathrm{def}}{=} (\forall n.\ (\mathsf{from\_signal}\ out\ n = \mathsf{Def}\ C)) \vee\\
&\quad (\exists m.\ \forall n.\ (\mathsf{from\_signal}\ out\ n = \mathsf{if}\ n < m\ \mathsf{then}\ \mathsf{Def}\ C\ \mathsf{else}\ \perp))\\
&\| \,\mathsf{Sink}\, \| \,(in : \alpha\ \mathsf{signal}) \stackrel{\mathrm{def}}{=} \mathsf{T}
\end{aligned}$$

Automated Program Synthesis for Java Programming Language*

Mait Harf, Kristiina Kindel, Vahur Kotkas, Peep Küngas, and Enn Tyugu

Department of Computer Science,
Institute of Cybernetics at Tallinn Technical University, Estonia
e-mail: {mait,kristina,vahur,peep,tyugu}@cs.ioc.ee

Abstract. In this paper we introduce a methodology of program synthesis for Java programming language by extending Java classes with high level specifications. The specifications are handled by a synthesizer also briefly described in this paper.

1 Introduction

As the number of software components grows exponentially, software designers are unable to manage all software libraries. To overcome this problem, we should use automated software design, where libraries are handled automatically with the help of specifications provided by a designer.

One way to automate the software design process is to use Structural Synthesis of Programs (SSP) [1]. SSP is a technique of deductive synthesis of programs based on automatic proof search in intuitionistic propositional calculus, where the solving complexity is hidden from the end-user into the system.

The idea of using SSP for automated program generation is not a new one. Already in the seventies the Priz family of programming languages was developed in the Institute of Cybernetics, that allowed engineers to solve their tasks using a very high-level programming language. A similar approach has been successfully used in the Amphion system [2].

In this paper we introduce a methodology of extending Java classes with capabilities of SSP. The Java language has been chosen as the platform for SSP due to its relative robustness and flexibility. Our ultimate goal is to increase the programming efficiency through the use of general solvers for variety of problems and software reuse.

This work is inspired by the research done at the Institute of Cybernetics, Estonia, for several decades and related to the work of Sven Lämmerman [3] from Royal Institute of Technology, Sweden.

2 An Example of the Specification Language

In the current section we use a modeling of radar coverage as an example. When starting to model something, we usually take a handbook of the field of interest and study it. When the area of interest is radar performance calculation the main equation that can be found is:

$$R^4 = \frac{P_t G^2 \lambda^2 \sigma}{(4\pi)^3 SNR_{\min} LN}, \tag{1}$$

where P_t is transmit power, G is antenna gain, λ is wavelength, σ is target radar cross section, $SNR_{\min}$ is minimal signal to noise ratio for target detection with certain probabilities, L is signal losses, N is total received noise and R is detection range.

To start modeling we create a new Java class called *Radar*, declare the listed components and add a declarative specification to the Java class that describes the relations among the components. The declarative specification is added to the Java class as a string array.

```
"var r, wavelen : Length" // detection range, wavelength
"var pt, prf, rcst : Integer" // power, pulse rep. freq., target size
"var g : Gain" // antenna power gain
"var snr_min : Ratio" // minimal Signal to Noise Ratio
"var losses : Losses" // system losses
"var noise : Noise" // total noise power
"rel r^4 = pt*g^2*wavelen^2*rcst / ((4*pi)^3*snr_min*losses*noise))"
```

* This work was partially funded by Estonian Innovation Foundation under the contract No. 6kl/00.

```
"rel snr_min.prf == prf"
"rel noise.r == r"
"rel noise.prf == prf"
```

In this specification we have to redefine all the components of the Java class (statements starting with *var*), that would be used in the relations (statements starting with *rel*).

Three types of relations can be described in specifications: equations and equivalences (see example specification above) and Java method declarations (see example of class *RadarModel*).

```
public class RadarModel implements SSPinterface {
    public static String[] SSPspec = { "var radar : Radar",
        "var ver_cov : Coord []",
        "rel [radar.hor_ang, radar.ver_ang -> radar.r]"
            + " -> ver_cov {CalcVerCov}"};
    Radar radar;
    Coord[] ver_cov;
    public void RadarModel() {
        radar = new Radar();
        radar.wavelen = new Length(3.25, "cm");
    /* ... other initializations in the same way ...*/ }

    public static void main(String[] args) {
        RadarModel model = new RadarModel();
        SSP.compute(model, "ver_cov"); }
    public Coord[] CalcVerCov() { /*method body follows here*/ } }
```

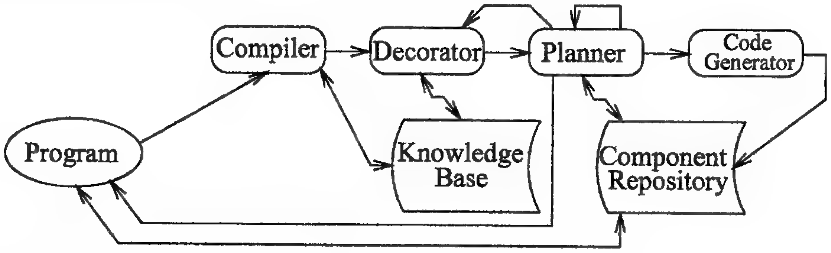
In the class *RadarModel* we create an instance of the class *Radar*, evaluate some of its components (see the constructor in the class *RadarModel*) and invoke the method *compute* of class *SSP*. The method *compute* runs the synthesizer, which builds a new class containing a method for computing the required component *ver_cov*.

In the class *RadarModel* we have a relation that specifies how to calculate the radar vertical coverage. For vertical coverage calculation a method *CalcVerCov* can be used, if a subtask that calculates component r of *radar* from given vertical and horizontal angle and other components that were evaluated before is solvable. Using this subtask in a loop for calculating the maximal detection ranges for several elevation angles the method computes the vertical coverage diagram.

Similar methodology has been used in the radar coverage modeling package in a programming environment NUT [4]. Experiments show that the methodology is suitable for solving problems of this kind.

3 The Synthesizer

We propose an architecture of the synthesizer, which is decomposed into 6 logically separated components: Knowledge Base (KB), Compiler, Decorator, Planner, Code Generator and Component Repository. The interconnections between these components are presented on the figure below.



The task of the Compiler is to parse declarative specifications and store the results into a KB. The KB represents a memory system designed to hold data structures being used later for planning structure creation.

The Decorator creates a special structure suitable for fast planning, sets evaluation states of components on the created structure and passes it all to the Planner.

The Planner is used for automatic program synthesis from problem description. The SSP principles and the proof search algorithms proposed in [1, 5] are applied in the Planner.

The problem description is of the form $x \longrightarrow y$, where x denotes the set of known components and y denotes the set of components to be computed. The task of the Planner is to construct an algorithm (a sequence of relations) that describes how to compute y from x.

Before starting the problem solving the Planner checks from the Component Repository whether a solution already exists for it or not. In the case the solution does not exist, the Planner starts solving it. The proof search strategy of SSP applied in the Planner is:

- an assumption-driven forward search to build a sequence of relations to be applied for solving the problem. The search is a simple flow analysis on the network of relations.
- a goal-driven backward search to solve subtasks.
- a minimization of the synthesized algorithm applied to the synthesized algorithm.

As a result of planning we get an algorithm that is not necessarily the shortest, however it does not contain unnecessary relations.

If the solution is found, class code is generated by the Code Generator. The code is then compiled to Java class and added to the Component Repository.

The aim of Component Repository is to maintain a set of components solving a certain problem. The components can be reused to solve similar tasks when the problem description is matching.

4 Summary

In the current paper we proposed a methodology of structural program synthesis for Java programming language by extending Java classes with high level specifications. Our main objective is to create a tool that supports simulation, prototyping, software reuse and increases programming efficiency.

References

1. E. Tyugu. The structural synthesis of programs. Lecture Notes in Computer Sciences, Vol. 122, 1981, pp. 290–303.
2. M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, I. Underwood. Deductive Composition of Astronomical Software from Subroutine Libraries. In: 12th Conference on Automated Deduction. A. Bundy, (ed). Springer-Verlag Lecture Notes in Computer Science, Vol. 814, 1994.
3. S. Lämmermann. Automated Composition of Java Software, licentiate thesis, Department of Teleinformatics, Royal Institute of Technology, Sweden, May 2000.
4. J. Penjam et al. Ontology-based design of surveillance systems with NUT. Proceedings of the Third International Conference on Information Fusion, Paris, Vol. 2, 2000.
5. M. Harf, E. Tyugu. Algorithms of structured synthesis of programs. Programming and Computer Software, Vol. 6, 1980, pp. 165–175.

Semantics & Types

What Use Is Formal Semantics?

Peter D. Mosses

BRICS & Department of Computer Science, University of Aarhus
Ny Munkegade bldg. 540, DK-8000 Aarhus C, Denmark
e-mail: pdmosses@brics.dk, Web: http://www.brics.dk/~pdm

Abstract. Formal descriptions of syntax are quite popular: regular and context-free grammars have become accepted as useful for documenting the syntax of programming languages, as well as for generating efficient parsers; regular expressions are extensively used for searching and transforming text. Formal semantic descriptions, in contrast, are widely regarded as only of academic interest, and they have so far found little application in practical software development.

In this paper, we survey the main frameworks for formal semantics: operational, denotational, and axiomatic semantics, together with some more recent hybrid approaches. We assess the potential and actual use of descriptions in the various frameworks, and consider also their practical aspects, such as comprehensibility, modularity, and extensibility, which are especially significant when describing full-scale languages. Finally, we argue that formal semantics will never be regarded as truly useful until tools become available for transforming well-engineered semantic descriptions into efficient compilers and interpreters.

The paper is intended to be accessible to all computer scientists. Familiarity with the details of particular semantic frameworks is not required, although an understanding of the general idea of formal semantics is assumed.

1 Introduction

▷ Formal *syntax* is *widely used* in practical applications.

Formal syntax includes various frameworks for specifying languages as set of strings: regular expressions, and several kinds of grammar. Regular expressions and regular grammars originated from theoretical studies of automata and the languages that they accept. Regular expressions have become widely adopted by programmers, since they enable efficient search for (and replacement of) particular patterns in programs and other texts. Tools such as `lex` and `yacc` generate scanners and parsers from regular, resp. LALR(1) context-free grammars; more sophisticated tools such as the CWI Meta-environment [11] are even able to generate efficient parsers from unrestricted context-free grammars. Attribute grammars of various kinds are used in compiler-writing systems to generate type-checkers and code-generators.

▷ Formal *semantics* is *almost never used* in practical applications.

In marked contrast to formal syntax, formal semantics has so far hardly ever been exploited in practical applications by programmers—not even in connection with compiler-writing. This cannot be due to a shortage of semantic frameworks to choose from: as we shall see in Sect. 2, quite a large number of distinct frameworks have been developed. Nor has there been a lack of theoretical effort in establishing the foundations of the various frameworks. The major semantic frameworks have also been quite widely taught (even at the undergraduate level) and plenty of pedagogical text-books are available. The potential benefits of using formal semantics in general, as well as the special advantages of particular frameworks, have been proclaimed in numerous books and papers. Yet despite all this investment of effort in promoting formal semantics, significant practical uses of it remain few and far between.

▷ We shall survey the various semantic frameworks, list some significant uses, and speculate on the hindrances for greater use.

Our survey of semantic frameworks in Sect. 2 might appear very superficial. However, it is hoped that it may still be quite useful, since leaving out the details draws attention to the most important differences—and similarities—between the frameworks.

The list of significant uses of formal semantics in Sect. 3 may have unjustly omitted some important examples known to the reader—the author would be grateful to receive references to the relevant publications in such cases. Note however that case studies are not deemed to be relevant here when their only purpose or result was merely to demonstrate that some framework could indeed be used to describe some language.

Regarding the author's speculations on hindrances in Sect. 4, it may well be that factors outside the scope of this paper are in any case dominant in inhibiting the use of formal semantics (e.g., the short-term nature of budgeting in conventional software development projects).

▷ Our conclusion will be that wider use of formal semantics depends on the possibility of generating *efficient* implementations from *well-engineered* semantic descriptions

Some frameworks may appear to be inherently better-suited for generating efficient implementations. For instance, it might be imagined that operational semantics would have decisive advantages over the more abstract denotational and axiomatic approaches. However, any kind of semantics is supposed to *determine* the observable behaviour of all programs in the described language; provided that the relevant information about behaviour can be extracted automatically from the semantics, the generation of implementations from that information may be largely independent of how it was originally presented. In general, efficiency of the generation process itself is nothing like as important as efficiency of running programs via the generated implementation.

Currently, very few systems generating *any* kind of implementation from semantic descriptions are available [21]. Most of the semantics-directed compiler and interpreter generators reported in the literature were developed in the 1980's, and the languages and compilers used to implement them have changed so much in the meantime that it would often take a major investment of effort to get the systems running again. Moreover, the reported performance figures are sometimes difficult to interpret relative to present-day machines.

However, the use of even a properly-implemented and well-maintained system is limited to the *input* actually available for it. In the case of semantics-directed compiler generators, the input should be *complete, fully formal* semantic descriptions—in contrast to most semantic descriptions found in textbooks and papers, where "obvious" details may be left to the reader, and semi-formal "conventions" are often introduced in the interests of conciseness.

The author painfully recalls how, after having invested considerable effort over a number of years in developing SIS (the first semantics implementation system for denotational semantics [35], see also [50,55]), he came to the realization that, regardless of how attractive the semantic framework might be for theoretical reasons, its poor "semantics engineering" aspects made it simply too tedious and error-prone to provide complete descriptions of larger languages as input for the system. (The subsequent investigation of how one might improve the engineering aspects of denotational semantics led to the development of action semantics, which is surveyed Sect. 2.5.)

▷ For a quick first reading, focus on these main points and skip the intervening text.

It is hoped that such display of the main points does not significantly hinder a continuous reading of the text.

2 Formal Semantics

▷ Most frameworks for (dynamic) semantics can be classified as *operational*, *denotational*, or *axiomatic*.

Here, we focus entirely on *dynamic* semantics, which concerns the run-time behaviour of programs. Static semantics addresses compile-time issues such as type-checking, and it is an essential ingredient in complete language descriptions, but unfortunately outside the scope of the present paper.

We survey the main frameworks that have been developed within each of the main classifications. A few frameworks do not fall into the primary classifications, being essentially *hybrids* of different kinds of frameworks. In fact, some frameworks that are traditionally classified as operational might better be regarded as hybrid, since they involve some distinctive features of denotational semantics.

▷ Most frameworks are based on *context-free abstract syntax.*

Use of *abstract* syntax implies that semantics is defined for *trees* that represent the "deep" structure of programs. The concrete syntax issues of unambiguously parsing a program text so as to discover its structure are thus of no concern in semantic descriptions. Of course a complete language description has then to specify exact the relationship between concrete and abstract syntax, but that is usually completely straightforward.

Context-free abstract syntax is particularly pleasant to work with, and has clean and simple algebraic foundations. The more complex alternatives are to use higher-order abstract syntax (which pre-supposes static scopes for bindings) or some form of attributed syntax (which may undermine compositionality of semantics).

In many semantic frameworks, abstract syntax is specified by ordinary BNF-like grammars, exploiting keywords and symbols from concrete syntax to distinguish between alternative abstract constructs. This makes it easy to guess the intended relationship between concrete and abstract syntax, and facilitates the reading of semantic descriptions. However, such abstract syntax grammars are always interpreted as defining sets of *trees*, rather than sets of strings. They tend to be significantly simpler than unambiguous context-free grammars for concrete syntax.

▷ We shall illustrate the various semantic frameworks with fragments involving the description of a simple if-statement and a comparative expression.

An abstract syntax for if-statements, empty statements, and numerical comparison expressions is specified in Table 1. We do not try to make a syntactic distinction between boolean and numerical expressions, since the types of identifiers in expressions would usually depend on their declarations. When formulating dynamic semantics, one need not worry about what semantics is given to programs containing ill-typed constructs, since such programs should be filtered out by a preceding static semantics.

Notice that the grammar given for expressions would be ambiguous in concrete syntax, whereas in abstract syntax the alternative *Exp* >= *Exp* simply specifies that both the left and right sub-expressions are of the same sort *Exp*. (Incidentally, some authors prefer to use possibly-subscripted variables, rather than sort names, as nonterminal symbols in abstract syntax grammars.)

For later use, expressions are assumed to compute values $v \in V$, including integers $n \in Z$ and the boolean values $tt, ff \in B$. (Statements are assumed to compute a fixed null value.)

Table 1. Abstract Syntax

$$\begin{array}{l} s \in \mathit{Stm} ::= \mathtt{if}\,(\mathit{Exp})\,\mathit{Stm} \mid \{\ \} \mid \ldots \\ e \in \mathit{Exp} ::= \mathit{Exp} \mathbin{\mathtt{>=}} \mathit{Exp} \mid \ldots \end{array}$$

▷ Formal semantics aims at *modelling* the *observable behaviour* of *complete programs.*

Low-level implementation-dependent details are generally ignored, e.g., finite bounds on the size of numbers, arrays, and recursive procedure calls, or recycling of storage cells. Auxiliary entities, regardless of whether they are related to anticipated features of implementations, may be introduced ad libitum: e.g. environments $\rho \in Env$, stores $\sigma \in S$. Some frameworks model also the contributions of all parts of programs to overall behaviour, but this is not obligatory.

2.1 Operational Semantics

▷ Operational semantics models the *computations* of programs.

A computation is usually regarded as a (perhaps infinite) *sequence of steps* between states. An alternative approach is to reflect the inductive structure of programs and represent computations as *derivation trees*, where the steps occur as leaves but their order is left open.

▷ The semantics of a program is determined by the set of possible computations, modulo some equivalence relation.

States in computations generally incorporate the abstract syntax of the entire program—or at least, the part of it that remains to be executed. Thus no two syntactically-distinct programs can ever have the same (non-empty) sets of possible computations, even when their differences are obviously insignificant. To obtain a reasonable notion of semantic equivalence in operational semantics, some equivalence relation that ignores the syntactic components of states has to be introduced; a popular choice is bisimulation equivalence [30].

Structural Operational Semantics (SOS) was proposed by Plotkin in 1981 [46], and the basic ideas have since been presented (on a less ambitious scale) in various textbooks (e.g. [44]), and exploited in numerous papers on concurrency [29].

▷ Computations are modelled as sequences of (labelled) transitions between states involving syntax, computed values, and auxiliary entities.

The sequences may be finite or infinite. The states themselves are finite mathematical entities, but in contrast to automata theory, the sets of states here are generally infinite.

Characteristic for SOS is that as a computation proceeds, parts of the program tree are gradually replaced by the values that they have computed; it may also happen that parts get replaced by different trees, possibly involving auxiliary constructs not present in the language being described. Auxiliary components of states may include stores and environments. The latter are however usually taken as an extra argument of the transition relation; they can be avoided altogether by use of syntactic substitution.[1]

▷ Transitions are specified by axioms and inference rules.

In sequential languages, a step for a compound phrase depends on a step for a sub-phrase, whereas in concurrent languages, it may depend on synchronized steps for more than one phrase. See Table 2 for an SOS for the fragments whose abstract syntax was specified in Table 1. Notice that it is specified that the evaluation of e_2 can only proceed after e_1 has computed a number n_1, but it would be just as easy to allow interleaving (simply by replacing n_1 in the conclusion of the inference rule by e_1).

Table 2. Structural Operational Semantics

$$\boxed{s,\sigma \longrightarrow s',\sigma'}$$

$$\frac{e,\sigma \longrightarrow e',\sigma'}{\mathtt{if}\,(e)\,s,\sigma \longrightarrow \mathtt{if}\,(e')\,s,\sigma'} \qquad \begin{array}{l} \mathtt{if}\,(tt)\,s,\sigma \longrightarrow s,\sigma \\ \mathtt{if}\,(f\!f)\,s,\sigma \longrightarrow \{\},\sigma \end{array} \tag{1}$$

$$\boxed{e,\sigma \longrightarrow e',\sigma'}$$

$$\frac{e_1,\sigma \longrightarrow e_1',\sigma'}{e_1 \mathtt{>=} e_2,\sigma \longrightarrow e_1' \mathtt{>=} e_2,\sigma'} \qquad \frac{e_2,\sigma \longrightarrow e_2',\sigma'}{n_1 \mathtt{>=} e_2,\sigma \longrightarrow n_1 \mathtt{>=} e_2',\sigma'} \tag{2}$$

$$n_1 \mathtt{>=} n_2,\sigma \longrightarrow tt,\sigma \text{ if } n_1 \geq n_2 \qquad n_1 \mathtt{>=} n_2,\sigma \longrightarrow f\!f,\sigma \text{ if } n_1 < n_2 \tag{3}$$

Natural Semantics was developed by Kahn and his group at Sophia Antipolis in the mid-1980's [26]. It is sometimes referred to as "big-step" SOS. It can be used together with the pure "small-step" SOS; Plotkin achieved the same effect by letting a small step of statement execution involve a *series* of small steps for expression evaluation, formally specified using transitive closure.

▷ Terminating computations are modelled as *proof trees* for evaluation relations between syntax and computed values, possibly depending on auxiliary entities.

[1] The actual definition of substitution is usually left to the reader...

One major drawback of natural semantics is that nonterminating computations are ignored. Notice that computed values do not need to be allowed as components of abstract syntax trees in natural semantics, in contrast with SOS.

The usual style is to exhibit the environment as an extra argument to the evaluation relation, as illustrated in Table 3; the resemblance to sequents in Gentzen calculi led to the name of the framework.

▷ Evaluations are specified by axioms and inference rules.

In sequential languages, evaluation of a compound phrase depends on the evaluation of all the involved subphrases. However, the rules are not strictly inductive, in general: e.g., a (terminating) evaluation of a loop may depend on another evaluation for the same loop. The description of concurrent languages, or even of interleaved expression evaluation, is problematic in natural semantics. The need to "thread" effects on stores explicitly through premises of rules when describing conventional imperative languages is a considerable disadvantage of Natural Semantics—so much so that when it was adopted for the Definition of Standard ML [31], a convention was introduced so that the store could actually be left implicit in most rules (provided that the premises are written in the intended order of evaluation of sub-expressions).

Table 3. Natural Semantics

$$\boxed{s,\sigma \longrightarrow \sigma'}$$

$$\frac{\rho \vdash e,\sigma \longrightarrow tt,\sigma' \quad \rho \vdash s,\sigma' \longrightarrow \sigma''}{\rho \vdash \texttt{if(}e\texttt{)}s,\sigma \longrightarrow \sigma''} \qquad \frac{\rho \vdash e,\sigma \longrightarrow f\!f,\sigma'}{\rho \vdash \texttt{if(}e\texttt{)}s,\sigma \longrightarrow \sigma'} \tag{4}$$

$$\boxed{e,\sigma \longrightarrow v,\sigma'}$$

$$\frac{\rho \vdash e_1,\sigma \longrightarrow n_1,\sigma' \quad \rho \vdash e_2,\sigma' \longrightarrow n_2,\sigma''}{\rho \vdash e_1 \texttt{>=} e_2,\sigma \longrightarrow tt,\sigma''} \text{ if } n_1 \geq n_2 \tag{5}$$

$$\frac{\rho \vdash e_1,\sigma \longrightarrow n_1,\sigma' \quad \rho \vdash e_2,\sigma' \longrightarrow n_2,\sigma''}{\rho \vdash e_1 \texttt{>=} e_2,\sigma \longrightarrow f\!f,\sigma''} \text{ if } n_1 < n_2 \tag{6}$$

Reduction Semantics was developed by Felleisen and his colleagues towards the end of the 1980's [14].

▷ Computations are modelled as sequences of term rewriting steps (reductions).

Here, both computed values and auxiliary entities are represented as terms: there is little or no separation between syntactic and semantic entities. The sequences of rewriting steps may be infinite.

▷ Reductions are restricted to occur in evaluation contexts, the form of which is specified by a context-free grammar.

Observe that each alternative in the grammar for the contexts *STM* and *EXP* in Table 4 corresponds to an inference rule for the same construct in SOS. Clearly, it is more concise to specify a grammar than a set of inference rules. Moreover, reduction semantics has the advantage over SOS that a reduction step may replace the *context* as well as its contents, which can exploited not only to deal with effects on storage, but also with control constructs such as call/cc. Reduction semantics has the advantage over natural semantics that it can cope with non-terminating computations, as well as with synchronization and interleaving [48]; it does not appear to have significant advantages over SOS, apart from greater conciseness.

Table 4. Reduction Semantics

$$\begin{aligned} e \in \mathit{Exp} &::= \mathtt{true} \mid \mathtt{false} \mid \ldots \\ S \in \mathit{STM} &::= [\,] \mid \mathtt{if}\,(\mathit{EXP})\,\mathit{Stm} \mid \ldots \\ E \in \mathit{EXP} &::= [\,] \mid \mathit{EXP} \mathbin{\mathtt{>=}} \mathit{Exp} \mid Z \mathbin{\mathtt{>=}} \mathit{EXP} \mid \ldots \end{aligned}$$

$$\boxed{s \rightarrow s'}$$

$$\mathtt{if}\,(\mathtt{true})\,s \rightarrow s \qquad \mathtt{if}\,(\mathtt{false})\,s \rightarrow \{\} \tag{7}$$

$$\boxed{e \rightarrow e'}$$

$$n_1 \mathbin{\mathtt{>=}} n_2 \rightarrow \mathtt{true} \text{ if } n_1 \geq n_2 \qquad n_1 \mathbin{\mathtt{>=}} n_2 \rightarrow \mathtt{false} \text{ if } n_1 < n_2 \tag{8}$$

$$\boxed{S, \sigma \longrightarrow S', \sigma'}$$

$$S[e], \sigma \xrightarrow{S} [e'], \sigma \text{ if } e \rightarrow e' \tag{9}$$

$$S[s], \sigma \xrightarrow{S} [s'], \sigma \text{ if } s \rightarrow s' \tag{10}$$

Modular Operational Semantics was developed by the present author at the end of the 1990's [40,42], with the aim of improving some of the pragmatic aspects of the conventional SOS and natural semantics frameworks—inspired by the improvements obtained by the use of monads in denotational semantics, see Sect. 2.2.

▷ Modular SOS is a variant of SOS where states are restricted to syntax and computed values, and all auxiliary entities are incorporated in *labels* on transitions.

Labels in modular SOS may include e.g. environments, pairs of stores, and communication signals. The set of labels is generally infinite. It is straightforward to reduce a modular SOS to a conventional SOS, by moving the environments and stores back to their usual places.

▷ Computations require adjacent labels to be *composable.*

Composition of labels is usually partial: when labels contain environments, they compose only when the environments are the same, and when they contain pairs of stores, the second store in the first label has to be the same as the first store in the second label.

In fact the set of labels always forms a *category.* Let the variable α range over all labels, but let ι range only over *identity* labels. The objects of the label category may be regarded as the semantic components of states—which are clearly separated from the syntactic components in this framework, in contrast with all other operational frameworks. Notice in Table 5 how arbitrary labels α are propagated during the evaluation of sub-expressions, whereas labels on reduction steps are required to be identities ι.

New components can be added to labels by applying *label transformers,* which form product categories. Not only do label transformers leave the rules well-formed (so that they never need reformulating when extending the described language), but also computations and bisimulation equivalence are preserved [39].

▷ Similarly, Modular Natural Semantics requires all auxiliary entities to be incorporated in labels on evaluations.

Apart from the usual differences between SOS and natural semantics, observe in Table 6 that composition of labels has to be used explicitly in modular natural semantics. This composition replaces the threading of the store through premises of rules in conventional natural semantics, illustrated in Table 3. In fact sequential composition is not the only possibility for combining labels: when the labels of a modular natural semantics are taken to be finite sequences, it is possible to describe interleaving in terms of *shuffling* labels [39].

Abstract State Machines (ASMs), previously called "evolving algebras", is a framework proposed by Gurevich in the early 1990's [16].

Table 5. Modular Structural Operational Semantics

$$\boxed{s \xrightarrow{\alpha} s'}$$

$$\frac{e \xrightarrow{\alpha} e'}{\mathtt{if}\,(e)\,s \xrightarrow{\alpha} \mathtt{if}\,(e')\,s} \qquad \begin{array}{l}\mathtt{if}\,(tt)\,s \xrightarrow{\iota} s \\ \mathtt{if}\,(f\!f)\,s \xrightarrow{\iota} \{\}\end{array} \tag{11}$$

$$\boxed{e \xrightarrow{\alpha} e'}$$

$$\frac{e_1 \xrightarrow{\alpha} e_1'}{e_1 \mathrel{>=} e_2 \xrightarrow{\alpha} e_1' \mathrel{>=} e_2} \qquad \frac{e_2 \xrightarrow{\alpha} e_2'}{n_1 \mathrel{>=} e_2 \xrightarrow{\alpha} n_1 \mathrel{>=} e_2'} \tag{12}$$

$$n_1 \mathrel{>=} n_2 \xrightarrow{\iota} tt \text{ if } n_1 \geq n_2 \qquad n_1 \mathrel{>=} n_2 \xrightarrow{\iota} f\!f \text{ if } n_1 < n_2 \tag{13}$$

Table 6. Modular Natural Semantics

$$\boxed{s \xrightarrow{\alpha} \{\}}$$

$$\frac{e \xrightarrow{\alpha_1} tt \quad s \xrightarrow{\alpha_2} \{\}}{\mathtt{if}\,(e)\,s \xrightarrow{\alpha} \{\}} \text{ if } \alpha = \alpha_1 \,;\, \alpha_2 \qquad \frac{e \xrightarrow{\alpha} f\!f}{\mathtt{if}\,(e)\,s \xrightarrow{\alpha} \{\}} \tag{14}$$

$$\boxed{e \xrightarrow{\alpha} v}$$

$$\frac{e_1 \xrightarrow{\alpha_1} n_1 \quad e_2 \xrightarrow{\alpha_2} n_2'}{e_1 \mathrel{>=} e_2 \xrightarrow{\alpha} tt} \text{ if } n_1 \geq n_2 \land \alpha = \alpha_1 \,;\, \alpha_2 \tag{15}$$

$$\frac{e_1 \xrightarrow{\alpha_1} n_1 \quad e_2 \xrightarrow{\alpha_2} n_2'}{e_1 \mathrel{>=} e_2 \xrightarrow{\alpha} f\!f} \text{ if } n_1 < n_2 \land \alpha = \alpha_1 \,;\, \alpha_2 \tag{16}$$

▷ Computations are modelled as sequences of parallel sets of assignments to values of particular functions on particular arguments.

The sequences may be finite or infinite. Dynamic function values remain stable when not updated; functions declared to be static never get updated after their initial definitions.

▷ States include control-flow graphs representing the entire program.

In the fragment shown in Table 7, the functions *fst*, *nxt* represents *normal* control flow between phrases. However, flow of control need not follow the structure of the program at all: in principle, the pointer *task* to the part of the program currently being executed can be moved arbitrarily. The control-flow graph itself is static, but computed values can be associated with nodes by a separate dynamic function, such as *val* in Table 7. Scopes of bindings are represented indirectly, by explicit stacking of values, rather than by using environments or syntactic substitution.

Other Operational Frameworks

▷ Various other operational frameworks have been developed.

These include translation to code for the SECD abstract machine [27] and the VDL abstract machine [58], the SMoLCS framework [4], a generalization G^∞SOS of operational and inductive semantics to infinitary systems [9], and the EOS framework [10]. Lack of space unfortunately precludes further discussion of these frameworks.

Table 7. Abstract State Machines

```
$task : Phrase$
$fst, nxt : Phrase \to Phrase$
$val : Exp \to V$

let $s' =$ if$(e)s$ in                                                      (17)
    $fst(s') = fst(e), nxt(e) = s', nxt(s) = nxt(s')$

if $task$ is if$(e)s$ then                                                  (18)
  if $val(e)$ then $task := fst(s)$
              else $task := nxt(task)$

let $e' = e_1$ >= $e_2$ in                                                  (19)
    $fst(e') = fst(e_1), nxt(e_1) = fst(e_2), nxt(e_2) = e'$

if $task$ is $e_1$ >= $e_2$ then                                            (20)
  if $val(e_1) \geq val(e_2)$ then $val(task) := tt$
                               else $val(task) := ff$
```

2.2 Denotational Semantics

▷ Denotational Semantics models each part of a program as its *denotation*, representing its contribution to the overall behaviour of the enclosing program.

Denotations are usually higher-order functions between Scott-domains. The semantics of a complete program is its observable behaviour, which is obtained from its denotation.

▷ Semantic functions that map phrases to their denotations are defined inductively by sets of semantic equations, ensuring compositionality.

Such inductive definitions correspond to so-called initial algebra semantics. The semantics of loops and recursion involves explicit fixed-point operators, although some authors prefer to write these as equations whose (least) solution is to be found.

Scott-Strachey Semantics The Scott-Strachey style of denotational semantics is the original one, developed in Oxford at the end of the 1960's [36,44,50,52,54].

▷ Domains of denotations and auxiliary domains are defined by domain equations.

The domains are usually ω-complete partial orders (cpos), although lattices were used in the earliest papers; only *continuous* functions between domains are considered. Domain equations always have "least" solutions (up to isomorphism), e.g. $D = N + [D \to D]$ defines a domain D including both the natural numbers and all continuous functions on D. The elements of domains are specified in typed λ-notation.

▷ Typically, denotations are functions of environments, continuations, and stores.

Many standard techniques for representing programming concepts as pure mathematical functions have been established. For instance, sequencing may be represented either by composition of strict functions, or by use of *continuations*; the latter are illustrated in Table 8. The denotational description of nondeterminism, concurrency, and interleaving requires the use of power domains (and a significant amount of extra notation).

VDM Semantics The VDM style of denotational semantics was developed by Bjørner and C. B. Jones in the mid-1970's [5].

▷ The meta-notation Meta-IV used in VDM provides extra generality regarding abstract syntax.

Table 8. Scott-Strachey Semantics

$$
\begin{array}{ll}
\theta \in C = S \to A & \\
\kappa \in K = V \to C & \\
\mathcal{S} :\ \mathit{Stm} \to \mathit{Env} \to C \to C & \\
\mathcal{E} :\ \mathit{Exp} \to \mathit{Env} \to K \to C & \\
\mathcal{S}[\![\mathtt{if}\,(e)\,s]\!] = \lambda\rho.\lambda\theta.\mathcal{E}[\![e]\!]\rho(\lambda v.v|B \to \mathcal{S}[\![s]\!]\rho\theta, \theta) & (21) \\
\mathcal{E}[\![e_1 \mathtt{>=} e_2]\!] = \lambda\rho.\lambda\kappa.\mathcal{E}[\![e_1]\!]\rho(\lambda v_1.\mathcal{E}[\![e_2]\!]\rho(\lambda v_2.\kappa(v_1|Z \geq v_2|Z))) & (22)
\end{array}
$$

Lists, sets, and maps may be used as components of abstract syntax trees. This allows some semantic properties, such as the insignificance of the order of declarations in certain languages, to be made evident in the abstract syntax.

▷ It also ensures propagation of effects and exceptions through sequencing.

Meta-IV also provides notation for sequencing effects on stores, the use of which is illustrated in Table 9, and for exception-handling.

Table 9. VDM Semantics

$$
\begin{array}{ll}
\mathcal{M} : \mathit{Stm} \to (S \stackrel{\sim}{\to} S) & \\
\mathcal{M} : \mathit{Exp} \to (S \stackrel{\sim}{\to} S \times V) & \\
\mathcal{M}[\mathit{mk\text{-}If}(e, s)] = \text{def } v : \mathcal{M}[e]; & (23) \\
\qquad\qquad\qquad\quad \text{if } v \text{ then } \mathcal{M}[s] \text{ else } I_S & \\
\mathcal{M}[\mathit{mk\text{-}Boolinfixexpr}(e_1, GEQ, e_2)] = \text{def } v_1 : \mathcal{M}[e_1]; & (24) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{def } v_2 : \mathcal{M}[e_2]; & \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{return}(v_1|Z \geq v_2|Z) &
\end{array}
$$

Monadic Semantics The monadic style of denotational semantics was developed by Moggi at the end of the 1980's [32,33].

▷ Denotations in Monadic Semantics are elements of *monads*, and their composition is expressed independently of the domains used to construct the monads.

Various notations for monadic composition are available. In the one whose use is illustrated in Table 10, 'let $v = c_2$ in c_2' expresses that the computation c_1 is performed first; the computed value is then referred to as v in the computation c_2. In so-called exception monads, raising an exception in c_1 may cause c_2 to be skipped; various other monads define composition in different ways. The use of the composition notation depends only on knowing the type of value computed by c_1, not on the structure of the domain of computations in any particular monad. The notation '$[v]$' expresses the trivial computation that merely returns the value v. The closeness of the monadic notation to that provided a decade earlier by Meta-IV (see Table 9) is quite striking.

▷ Monad transformers construct monads *incrementally*, and *lift* the associated functions.

The order of applying the transformers can be critical—in particular, the transformers that provides continuations does not commute with other transformers. Moreover, not all functions can be lifted uniformly through monad transformers.

Moggi has subsequently developed a more general framework based on translation between meta-languages [34].

Table 10. Monadic Semantics

$$\mathcal{S} : \mathit{Stm} \to T()$$
$$\mathcal{E} : \mathit{Exp} \to T(V)$$
$$\begin{aligned}\mathcal{S}[\![\mathtt{if}\,(e)\,s]\!] = {} & \text{let } v = \mathcal{E}[\![e]\!] \text{ in} \\ & \text{case } v|B \text{ of } \mathit{tt} \Rightarrow \mathcal{S}[\![s]\!] \mid \mathit{ff} \Rightarrow [()]\end{aligned} \tag{25}$$
$$\begin{aligned}\mathcal{E}[\![e_1 \mathtt{>=} e_2]\!] = {} & \text{let } v_1 = \mathcal{E}[\![e_1]\!] \text{ in} \\ & \text{let } v_2 = \mathcal{E}[\![e_2]\!] \text{ in} \\ & [v_1|Z \geq v_2|Z]\end{aligned} \tag{26}$$

Predicate Transformer Semantics This approach, proposed by Dijkstra in the mid-1970's [12], is often regarded as axiomatic, since it involves assertions about the values of variables before and after executing statements. Here, however, it is classified as denotational.

▷ The denotation of a phrase is a predicate transformer that returns the weakest condition which ensures termination of the phrase with the argument condition holding.

Predicate transformers are required to have properties corresponding to continuity of functions on Scott-domains. The description of assignment involves substitution in formulae. Expression evaluation is assumed to be free of side-effects, errors, and non-termination, so that boolean expressions may be used as formulae, and numerical expressions substituted for variables. An advantage of the formalism is that non-determinism is straightforward to describe.

Table 11. Predicate Transformer Semantics

$$\mathit{wp} : \mathit{Stm} \times \mathit{Pred} \to \mathit{Pred}$$
$$\mathit{wp}(\mathtt{if}\,(e)\,s, Q) \Leftrightarrow (e \Rightarrow \mathit{wp}(s, Q)) \wedge (\neg\, e \Rightarrow Q) \tag{27}$$

Other Denotational Frameworks

▷ Various other denotational frameworks have been developed.

These include Naive Denotational Semantics [7], partially-additive semantics [28], use of metric spaces [1], and Extensible Denotational Semantics [8].

2.3 Axiomatic Semantics

▷ Axiomatic semantics restricts the potential models of phrases by asserting properties.

There may be more than one model of an axiomatic semantics, or there may be no models at all [2]. As with predicate transformers, boolean expressions are used as formulae, and the semantics of assignment statements involves substitution of numerical expressions for variables, so expressions cannot have side-effects, nor fail to return a value.

Hoare Logic was developed in the late 1960's [22].

▷ Partial correctness assertions $P\{s\}R$ require that if P holds and the subsequent execution of the statement s terminates, then R holds.

$P\{s\}R$ does not require s to terminate.

▷ The partial correctness assertions for statements are specified by axioms and inference rules.

General rules allow strengthening of pre-conditions and weakening of post-conditions.

Table 12. Hoare Logic

$$\frac{(P \wedge e)\{s\}R \qquad (P \wedge \neg e \Rightarrow R)}{P\{\mathtt{if}\,(e)\,s\}R} \tag{28}$$

Algebraic Semantics for programming languages was developed by Hoare and his colleagues at Oxford in the 1990's [23].

▷ Equations and inclusions between phrase terms characterize the relationship between their interpretations.

This approach can be applied directly only to languages that have an expressive syntax and clean semantics.

Table 13. Algebraic Semantics

$$\mathtt{if}\,(e)\,P = (e \rightarrow P \,[]\, \neg e \rightarrow \mathit{skip}) \tag{29}$$

$$e \rightarrow P = e^{\top};\, P \tag{30}$$

$$\ldots \tag{31}$$

Other Axiomatic Frameworks

▷ Not many axiomatic frameworks have been developed.

One notable one is Dynamic Logic [19].

2.4 Complementary Semantics

▷ The various semantic frameworks may be best suited for different uses.

It has been proposed several times [23,24,50] that it might be desirable to give descriptions of the same language in several frameworks

▷ Different descriptions of the same language have to be consistent.

It is well known that it is hard work to prove consistency between semantic descriptions in different formalisms. To derive one description from another, for example by use of abstract interpretation [9], may ensure consistency.

2.5 Hybrid Semantics

▷ A hybrid approach to semantics involves more than one framework in the same description.

The use of both regular and context-free grammars is an example of a hybrid approach in descriptions of syntax.

▷ The various semantic frameworks may have advantages for different stages of complete semantic descriptions.

The separation of semantics into static and dynamic phases encourages a hybrid approach to complete semantic descriptions; here we consider hybrid descriptions also within dynamic semantics.

Action Semantics was developed by the present author, in collaboration with Watt, in the second half of the 1980's [37,43,57].

▷ Action Semantics is a hybrid of denotational and operational semantics.

As in denotational semantics, inductively-defined semantic functions map phrases to their denotations, only here, the denotations are so-called *actions*; the notation for actions is itself defined operationally [37,41].

▷ It retains the idea of describing a programming language by reducing it to some known semantic universe.

Inductive definitions of semantic functions using semantic equations seem to be optimal.

▷ Action semantics avoids the use of higher-order functions expressed in lambda-notation.

The universe of pure mathematical functions is so distant from that of (most) programming languages that the representation of programming concepts in it is often excessively complex. The foundations of reflexive Scott-domains and higher-order functions are unfamiliar and inaccessible to many programmers. The use of pure lambda-notation has some pragmatic drawbacks; the monadic approach was developed (partly) to circumvent these.

▷ Action semantics provides a rich action notation with a direct operational interpretation.

The universe of actions involves not only control and data flow, but also scopes of bindings, effects on storage, and interactive processes, allowing a simple and direct representation of many programming concepts. The foundations of action notation involve SOS and algebraic specifications, which are both generally regarded as more accessible than domain theory.

Table 14. Action Semantics

$$\begin{array}{l} \mathit{execute}\ : \mathit{Stm} \rightarrow \mathit{action}[\mathit{completing} \mid \mathit{storing} \mid \mathit{diverging} \mid \ldots] \\ \mathit{evaluate} : \mathit{Exp} \rightarrow \mathit{action}[\mathit{giving\ a\ value} \mid \mathit{storing} \mid \ldots] \end{array}$$

$$\begin{array}{l} \mathit{execute}[\![\texttt{if(}e\texttt{)}\,s]\!] = \\ \qquad \mathit{evaluate}\ e\ \mathit{then} \\ \qquad \mid \mathit{check}(\mathit{it\ is\ true})\ \mathit{then\ execute}\ s\ \mathit{or}\ \mathit{check}(\mathit{it\ is\ false}) \end{array} \tag{32}$$

$$\begin{array}{l} \mathit{evaluate}[\![e_1 \texttt{>=} e_2]\!] = \\ \qquad \mid \mathit{evaluate}\ e_1\ \mathit{and\ then\ evaluate}\ e_2 \\ \qquad \mathit{then\ give\ not}(\mathit{the\ given\ number}\#1\ \mathit{is\ less\ than\ the\ given\ number}\#2) \end{array} \tag{33}$$

Other Hybrid Frameworks

▷ Various other hybrid frameworks have been developed.

As mentioned earlier, various operational frameworks such as VDL may be considered as hybrids. The other hybrid frameworks include Modular Denotational Semantics [15], Modular Monadic Action Semantics [56], Type-Theoretic Interpretation [20], and translation between meta-languages [34].

3 Potential and Actual Uses

▷ Formal semantics may be useful to language designers for recording design decisions during language development.

Scott-Strachey denotational semantics was used during the design of Ada [13] and of Scheme [47]. Natural semantics was used during the later stages of the design of Standard ML [31], and Hoare Logic similarly for Pascal [25]. However, formal semantics was used only to reveal irregularities and other deficiencies of designs that had already been largely completed, and not as a primary means of documenting decisions during the entire design process.

▷ Even when not used during language design, the semantics of a completed language design may be provided for reference—potentially even for standardization.

The operational SMoLCS framework was used to give a reference semantics for full Ada [3] (the original denotational semantics formulated during the Ada design [13] addressed only the sequential constructs, although a denotational semantics of full Ada has also been given in VDM framework [6]). A reference description of Prolog has been given in the ASM approach. Action semantics was chosen to describe the language ANDF [18].

▷ Parts of language implementations may be derived (manually or automatically) from its formal semantics.

The VDM semantics of Ada mentioned above was used in the systematic development of the DDC Ada compiler.

▷ Formal semantics may be used to explain language concepts to students and programmers.

Several text-books combine explanations of programming concepts with Scott-Strachey semantics, as did Strachey's own lecture notes [53]. Plotkin's lecture notes on SOS [46] show that this can also be done in a structural operational framework. Watt uses action semantics in his book on programming languages and semantics [57].

▷ The process of giving formal semantics may lead to new insight into programming concepts and open new research areas.

The prime example here is how Scott's search for a mathematical model for the untyped λ-calculus, which was being used with a primarily operational interpretation in Strachey's early work on denotational semantics, led to the development of domain theory. Denotational semantics has also influenced the design of ML and Scheme, and continuations, developed originally for use in describing the semantics of jumps to labels, have been incorporated in several programming languages.

▷ A formal semantics may allow verification of program properties, of the validity of program transformations, and of overall properties of the described language.

Hoare Logic is closely associated with program verification based on assertions of invariants in Pascal, and can be used to generate verification conditions. The algebraic semantics of occam [49] can be used to justify program transformations.

4 Hindrances to Greater Use

Compared to the amount of effort that has been devoted to the development of various semantic frameworks over more than three decades, and all the potential uses listed above, the list of actual uses may be considered as disappointing. Why has there not been far greater use of formal semantics in practice? Some—or perhaps even all—of the following factors may be relevant:

▷ Many frameworks for semantics are not particularly user-friendly.

In most frameworks, the familiar programming concepts underlying a described language (such as flow of control and scopes of bindings) are not indicated by fixed symbols, but rather get encoded in patterns of use of general notation. This reduces readability (at least until the relevant patterns have been learned). A related potential hindrance is when the details of mathematical foundations are reflected directly in semantic descriptions, since those foundations may seem inaccessible to many potential users [51].

Practical use of formal semantics involves descriptions of major programming languages. Few frameworks scale up smoothly from the tidy illustrative languages described in text-books and papers to languages such as C and Java—even Standard ML, a clean language designed by theoreticians, turned out to be a considerable challenge to describe accurately in Natural Semantics. Tool support could alleviate the problems of writing, checking, and navigating in large-scale descriptions, but mature tools are totally lacking for most frameworks, in marked contrast to the sophisticated programming environments available to programmers.

▷ Much effort is required to develop a complete semantic description, but the potential benefits are somewhat intangible.

For theoreticians, there is generally little academic reward for producing a semantic description of a full programming language, unless it can be published as a book. Merely overcoming pragmatic problems in practical applications of semantics is not of much theoretical interest. Practitioners involved with designing and implementing programming languages have to weigh the effort of using a formal semantics against how much practical advantage it gives. In any case, the market of potential users of a semantic description is currently a very minor one.

▷ In contrast, formal syntax has become quite popular.

One reason for the popularity of formal grammars may be that BNF—at least when extended with notation for iteration and optional phrases—is exceptionally user-friendly. The major relevant concepts (alternatives, sequencing, and iteration) are all expressed directly, and the notational variation concerning iteration, although irritating, does not impede reading and understanding grammars. BNF scales up smoothly to descriptions of larger languages. Moreover, practical use of formal syntax is strongly supported by tools, both for prototyping grammars (e.g., to check that a grammar is actually LALR(1)) and for generating useful parsers.

Denotational semantics espoused the idea of extending BNF to semantics, but the good pragmatic aspects of BNF have been sadly lacking in denotational descriptions (at least during the 1970's and 1980's, before the development of the monadic approach); most of the other frameworks surveyed in Sect. 2 suffer from similar problems with pragmatic aspects, especially regarding lack of tool support.

5 Conclusion

▷ A large number of semantic frameworks have been provided during the past three decades.

We have classified them mainly as operational, denotational, and axiomatic, regarding some frameworks as hybrids. To give complementary semantic descriptions of the same language in different frameworks would involve too much extra work. It seems preferable to exploit the best features of the various frameworks synergistically, in different parts of a single description.

▷ There are plenty of potential uses for formal semantics, but disappointingly few actual practical applications of real significance.

Language designers seldom use formal semantics at all during the design process—and then only in the final stages. Occasionally, a formal semantics of a language has been provided for reference purposes, and the implementors have referred to it for clarification; but otherwise, the impact of formal semantics has been limited to stimulation of research, with some spin-off in the form of new programming techniques such as continuations and monads.

▷ The main hindrances to greater use of formal semantics appear to be lack of user-friendliness, and lack of tool support.

Semantic descriptions in many frameworks have poor pragmatic aspects, for instance concerning modularity and the smoothness of scaling up to descriptions of larger languages. Quality tools are badly needed to assist the writing, checking, and reading of semantic descriptions.

▷ Development of a user-friendly semantic framework allowing generation of efficient compilers should encourage greater use.

Semantics-directed compiler generation was a popular topic for PhD theses in the 1980's and the first half of the 1990's. Despite the construction of some promising prototype systems, no tools for producing efficient compilers directly from semantic descriptions appear to be currently available.[2] To have such tools would not only provide a real incentive for writing semantic descriptions, they would also allow the empirical testing of whether the semantics really does define the intended language.

▷ Action semantics is a good candidate for such a framework.

Action semantics was designed with user-friendliness as first priority; it was chosen for use in the description of ANDF because of its good pragmatic qualities [17]. Substantial experience with prototype systems for compiler and interpreter generation based on action semantics has already been obtained. A new, significantly simplified version of action semantics is currently about to be released—simple enough to be taught at undergraduate level. Not many people are working on action semantics at present. Readers who might be interested in helping to produce useful tools for action semantics, especially compiler generators, are cordially invited to take a closer look at the framework from the materials available via the home page at `http://www.brics.dk/Projects/AS`, and to contact the author if wanting to help.

References

1. P. America, J. de Bakker, J. N. Kok, and J. Rutten. Denotational semantics of a parallel object-oriented language. *Information and Computation*, 1989. To appear.
2. E. A. Ashcroft, M. Clint, and C. A. R. Hoare. Remarks on 'Program proving: Jumps and functions, by M. Clint and C. A. R. Hoare'. *Acta Inf.*, 6:317–318, 1976.
3. E. Astesiano et al. The Ada challenge for new formal semantic techniques. In *Ada: Managing the Transition*, pages 239–248. Cambridge University Press, 1986.
4. E. Astesiano and G. Reggio. SMoLCS driven concurrent calculi. In *TAPSOFT'87, Proc. Int. Joint Conf. on Theory and Practice of Software Development, Pisa*, volume 249 of *LNCS*. Springer-Verlag, 1987.
5. D. Bjørner and C. B. Jones. *Formal Specification and Software Development.* Prentice-Hall, 1982.
6. D. Bjørner and O. N. Oest. *Towards a Formal Description of Ada*, volume 98 of *LNCS*. Springer-Verlag, 1980.
7. A. Blikle and A. Tarlecki. Naive denotational semantics. In *Information Processing 83, Proc. IFIP Congress 83.* North-Holland, 1983.
8. R. Cartwright and M. Felleisen. Extensible denotational semantics specifications. In *TACS'94, Proc. Symp. on Theoretical Aspects of Computer Software, Sendai, Japan*, volume 789 of *LNCS*, pages 244–272. Springer-Verlag, 1994.
9. P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *POPL'92, Proc. 19th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, Albuquerque, New Mexico*, pages 84–94, 1992.
10. P. Degano and C. Priami. Enhanced operational semantics. *ACM Computing Surveys*, 28(2):352–354, June 1996.
11. A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping*, volume 5 of *AMAST Series in Computing.* World Scientific, 1996.

[2] The RML system [45] is however able to produce efficient *interpreters* from (a special form of) natural semantics.

12. E. W. Dijkstra. Guarded commands, non-determinacy, and formal derivations of programs. *Commun. ACM*, 18:453–457, 1975.
13. V. Donzeau-Gouge, G. Kahn, B. Lang, et al. *Formal Definition of the Ada Programming Language, Preliminary Version.* INRIA, 1980.
14. M. Felleisen and D. P. Friedman. Control operators, the SECD machine, and the λ-calculus. In *Formal Description of Programming Concepts III, Proc. IFIP TC2 Working Conference, Gl. Avernæs, 1986*, pages 193–217. North-Holland, 1987.
15. J. A. Goguen and K. Parsaye-Ghomi. Algebraic denotational semantics using parameterized abstract modules. In J. Diaz and I. Ramos, editors, *Proc. Int. Coll. on Formalization of Programming Concepts, Peñiscola*, number 107 in LNCS. Springer-Verlag, 1981.
16. Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods.* Oxford University Press, 1995.
17. B. S. Hansen and J. Bundgaard. The role of the ANDF formal specification. Technical Report 202104/RPT/5, issue 2, DDC International A/S, Lundtoftevej 1C, DK–2800 Lyngby, Denmark, 1992.
18. B. S. Hansen and J. U. Toft. The formal specification of ANDF, an application of action semantics. In [38], pages 34–42, 1994.
19. D. Harel. Dynamic logic. In *Handbook of Philosophical Logic*, volume II. D. Reidel Publishing Company, 1984.
20. R. Harper and C. Stone. A type-theoretic interpretation of Standard ML. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Robin Milner Festschrifft.* MIT Press, 1998. To appear.
21. J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM SIGPLAN Notices*, Mar. 2000.
22. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, 1969.
23. C. A. R. Hoare and H. Jifeng. *Unifying Theories of Programming.* Prentice-Hall, 1998.
24. C. A. R. Hoare and P. E. Lauer. Consistent and complementary formal theories of the semantics of programming languages. *Act Inf.*, 3:135–153, 1974.
25. C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Inf.*, 2:335–355, 1973.
26. G. Kahn. Natural semantics. In *STACS'87, Proc. Symp. on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*, pages 22–39. Springer-Verlag, 1987.
27. P. J. Landin. A formal description of Algol60. In *Formal Language Description Languages for Computer Programming, Proc. IFIP TC2 Working Conference, 1964*, pages 266–294. IFIP, North-Holland, 1966.
28. E. G. Manes and M. A. Arbib. *Algebraic Approaches to Program Semantics.* Springer-Verlag, 1986.
29. R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
30. R. Milner. Operational and algebraic semantics of concurrent processes. In J. van Leeuwen, A. Meyer, M. Nivat, M. Paterson, and D. Perrin, editors, *Handbook of Theoretical Computer Science*, volume B, chapter 19. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.
31. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, 1997.
32. E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Computer Science Dept., University of Edinburgh, 1990.
33. E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.
34. E. Moggi. Metalanguages and applications. In *Semantics and Logics of Computation*, Publications of the Newton Institute. CUP, 1997.
35. P. D. Mosses. SIS, Semantics Implementation System: Reference manual and user guide. Tech. Mono. MD–30, Dept. of Computer Science, Univ. of Aarhus, 1979. Out of print.
36. P. D. Mosses. Denotational semantics. In *Handbook of Theoretical Computer Science*, volume B, chapter 11. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.
37. P. D. Mosses. *Action Semantics.* Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.
38. P. D. Mosses, editor. *Proc. 1st Intl. Workshop on Action Semantics, Edinburgh, 1994*, number NS-94-1 in BRICS Notes Series. BRICS, Dept. of Computer Science, Univ. of Aarhus, 1994. `http://www.brics.dk/NS/94/1`.
39. P. D. Mosses. Foundations of modular SOS. Research Series RS-99-54, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. `http://www.brics.dk/RS/99/54`; full version of [40].
40. P. D. Mosses. Foundations of Modular SOS (extended abstract). In *MFCS'99*, volume 1672 of *LNCS*, pages 70–80. Springer-Verlag, 1999. Full version available at `http://www.brics.dk/RS/99/54/`.
41. P. D. Mosses. A modular SOS for Action Notation (extended abstract). In *AS'99*, number NS-99-3 in Notes Series, pages 131–142, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. Full version available at `http://www.brics.dk/RS/99/56/`.
42. P. D. Mosses. A modular SOS for ML concurrency primitives. Research Series RS-99-57, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. `http://www.brics.dk/RS/99/57/`.
43. P. D. Mosses and D. A. Watt. The use of action semantics. In *Formal Description of Programming Concepts III, Proc. IFIP TC2 Working Conference, Gl. Avernæs, 1986*, pages 135–166. North-Holland, 1987.
44. H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction.* Wiley, Chichester, UK, 1992.

45. M. Pettersson. *Compiling Natual Semantics*, volume 1549 of *LNCS*. Springer-Verlag, 1999.
46. G. D. Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN–19, Dept. of Computer Science, Univ. of Aarhus, 1981.
47. J. Rees, W. Clinger, et al. The revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 21(12):37–79, 1986.
48. J. H. Reppy. CML: A higher-order concurrent language. In *Proc. SIGPLAN'91, Conf. on Prog. Lang. Design and Impl.*, pages 293–305. ACM, 1991.
49. A. W. Roscoe and C. A. R. Hoare. The laws of occam programming. *Theoretical Comput. Sci.*, 60(2):177–229, 1988.
50. D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn & Bacon, 1986.
51. D. A. Schmidt. On the need for a popular formal semantics. *ACM SIGPLAN Notices*, 32(1), 1997.
52. D. S. Scott and C. Strachey. Toward a mathematical semantics for computer languages. In *Proc. Symp. on Computers and Automata*, volume 21 of *Microwave Research Institute Symposia Series.* Polytechnic Institute of Brooklyn, 1971.
53. C. Strachey. Fundamental concepts of programming languages. *Higher-Order and Symbolic Computation*, 13(1/2):11–49, 2000.
54. R. D. Tennent. The denotational semantics of programming languages. *Commun. ACM*, 19:437–453, 1976.
55. M. Tofte. *Compiler Generators.* Springer-Verlag, 1990.
56. K. Wansbrough and J. Hamer. A modular monadic action semantics. In *Conference on Domain-Specific Languages*, pages 157–170. The USENIX Association, 1997.
57. D. A. Watt. *Programming Language Syntax and Semantics.* Prentice-Hall, 1991.
58. P. Wegner. The Vienna definition language. *ACM Comput. Surv.*, 4:5–63, 1972.

Binding-Time Analysis for Polymorphic Types

Rogardt Heldal and John Hughes

Chalmers University of Technology, S-41296 GÖTEBORG
e-mail: heldal@cs.chalmers.se, Web: www.cs.chalmers/~heldal

1 Introduction

Partial evaluation is by now a well-established technique for specialising programs [13], and practical tools have been implemented for a variety of programming languages [2, 1, 16, 5]. Our interest is in partial evaluation of modern typed functional languages, such as ML [18] or Haskell [14]. One of the key features of these languages is polymorphic typing [17], yet to date the impact of polymorphism on partial evaluation has not been studied. In this paper we explain how to extend an offline partial evaluator to handle a polymorphic language.

1.1 Background: Polymorphism

A *polymorphic function* is a function which may be applied to many different types of argument. In ML and Haskell, the types of such functions are expressed using a "forall" quantifier: for example, the well-known *map* function, which applies a function to every element of a list, has the type

$$map :: \forall \alpha_1, \alpha_2.(\alpha_1 \rightarrow \alpha_2) \rightarrow [\alpha_1] \rightarrow [\alpha_2]$$

meaning that for *any* types α_1 and α_2, *map* takes a function of type $\alpha_1 \rightarrow \alpha_2$ and a list of type $[\alpha_1]$, and produces a list of type $[\alpha_2]$. ($[\alpha]$ is our notation for a list type with elements of type α).

Polymorphic functions are heavily used in real functional programs. In particular, library functions are frequently polymorphic, since the types at which they will be needed are not known when the library is written. The standard library contains many polymorphic functions such as *map* and *foldr* (which takes a binary operator and its unit, and combines the elements of a list using the operator). These polymorphic functions greatly simplify programming, for example, the sum of a list of integers xs can be computed as *foldr* $(+)$ 0 xs, and the conjunction of a list of booleans bs can be computed as *foldr* $(\wedge)$ **true** bs.

1.2 Background: Partial Evaluation

A partial evaluator is a tool which takes a program and a *partially known* input, and performs operations in the program which depend only on the known parts, generating a specialised program which processes the remainder. For example, specialising *foldr* to the inputs *foldr* $(+)$ 0 $[x, y, z]$, where x, y and z are unknown, would generate the specialised program $x + y + z + 0$. Here the construction of the known list, and the recursion over it inside *foldr*, have been performed by the partial evaluator: only the actual computations of the sum of the unknown quantities remains in the specialised code.

Partial evaluators can be classified into *online* and *offline*. Online partial evaluators decide dynamically during specialisation which operations to perform, and which to build into the residual program: an operator is performed if its operands are known in that particular instance. An offline partial evaluator processes an *annotated* program, in which the annotations determine whether an operator is to be applied or not. Offline partial evaluators are generally more conservative, but simpler and more predictable; we focus on this type in this article.

As an example, we annotate the *power* function, which computes x^n, for specialisation with a known value for n. We annotate each operator with a *binding-time*, S (static) or D (dynamic), and we write function application explicitly as @ so that we can annotate it. Operators annotated static are performed during partial evaluation.

$$\begin{aligned} power\ n\ x = \ & \mathbf{if}^S\ n =^S 0 \\ & \mathbf{then}\ Int^{SD}\ 1 \\ & \mathbf{else}\ x \times power @^S (n -^S 1) @^S x \end{aligned}$$

In annotated programs we distinguish between known static values, and the corresponding dynamic code fragment; in this example, since the result of specialising *power* is code, the *coercion* Int^{SD} must be used to convert the static integer 1 to the correct type.

Annotated programs can be *interpreted* by a partial evaluator, or *compiled* into a *generating extension*. This is a program which, given the partially known input, generates a specialised version of the annotated program directly. The generating extension of this annotated *power* function is itself a recursive function, which computes the static operations directly, and generates code for the dynamic ones. Running the generating extension with the arguments 3 and "x" (a code fragment) produces the code fragment "$x \times x \times x \times 1$". Notice that, in the generating extension, a static integer and a dynamic integer are represented by different types: the former by an integer, and the latter by a code fragment — for example, an abstract syntax tree. Thus coercions do real work.

However, fixed annotations work poorly in large programs. Library functions in particular may be called in many contexts, with combinations of static and dynamic arguments which are unknown at the time the function definition is annotated. This motivates *polychronic* annotations[1] containing binding-time *variables*, which are passed as parameters to annotated functions [7]. Using polychronic annotations, we can annotate the *power* function as

$$\begin{aligned} \mathit{power}\ \beta_1\ \beta_2\ n\ x = {} & \mathbf{if}^{\beta_1}\ n =^{\beta_1} \mathit{Int}^{S\beta_1}\ 0 \\ & \mathbf{then}\ \mathit{Int}^{S(\beta_1 \sqcup \beta_2)}\ 1 \\ & \mathbf{else}\ x \times^{\beta_1 \sqcup \beta_2} \mathit{power}\ \beta_1\ \beta_2 @^S (n -^{\beta_1} \mathit{Int}^{S\beta_1}\ 1) @^S x \end{aligned}$$

where the least upper bound of two binding times is determined by $S \leq D$. This version can be specialised to any combination of known and unknown arguments, but binding-times must actually be computed and passed as parameters in the generating extension, increasing the cost of specialisation somewhat. Notice also that many more coercions are needed, now that the binding-times are no longer known a priori.

The binding-time behaviour of this function can be captured by a *binding-time type*,

$$\mathit{power} :: \forall \beta_1, \beta_2 . \mathit{Int}^{\beta_1} \to^S \mathit{Int}^{\beta_2} \to^S \mathit{Int}^{\beta_1 \sqcup \beta_2}$$

in which each type constructor is annotated to indicate whether the corresponding value is known. Program annotations can be generated by inferring these types using a binding-time type system. Types must always be *well-formed*, in the sense that no static type appears under a dynamic type constructor.

2 What About Polymorphism?

When we try to incorporate polymorphic functions into this framework, we immediately run into difficulties. Consider, for example, a possible binding-time type for the *map* function:

$$\mathit{map} :: \forall \alpha_1, \alpha_2 . \forall \beta_1, \beta_2 . (\alpha_1 \to^{\beta_1} \alpha_2) \to^S [\alpha_1]^{\beta_2} \to^S [\alpha_2]^{\beta_2}$$

But not every instantiation of this type is well-formed: if either β_1 or β_2 is D, then neither α_1 nor α_2 may be instantiated to a static type, since this would produce an ill-formed type containing a static type under a dynamic type constructor. To capture such dependencies between variables, we add *constraints* to our binding-time types, which all instantiations must satisfy. Writing $\beta \rhd \alpha$ for the constraint that if β is D, then α must be a dynamic type, we can give a correct type for *map* as

$$\begin{aligned} \mathit{map} :: \forall \alpha_1, \alpha_2 . \forall \beta_1, \beta_2 . (\beta_1 \rhd \alpha_1, \beta_1 \rhd \alpha_2, \beta_2 \rhd \alpha_1, \beta_2 \rhd \alpha_2) \Rightarrow & \\ (\alpha_1 \to^{\beta_1} \alpha_2) \to^S [\alpha_1]^{\beta_2} \to^S [\alpha_2]^{\beta_2} & \end{aligned}$$

These constraints have been used before [7], but did not appear in binding-time types since that paper did not consider polymorphism.

Now consider an even simpler polymorphic function,

$$\mathit{twice}\ f\ x = f @ (f @ x)$$

The standard type of this function is $\forall \alpha . (\alpha \to \alpha) \to \alpha \to \alpha$, but for the purposes of specialisation we can be more liberal: we can allow the argument and result of f to have different binding-time types, provided the result

[1] Also, confusingly, called "polymorphic".

can be coerced to the argument type. Thus we also need a *coercion* or *subtyping* constraint $\alpha_1 \leq \alpha_2$, which lets us give *twice* the binding-time type

$$\mathit{twice} :: \forall \alpha_1, \alpha_2.\forall \beta.(\beta \rhd \alpha_1, \beta \rhd \alpha_2, \alpha_2 \leq \alpha_1) \Rightarrow (\alpha_1 \rightarrow^{\beta} \alpha_2) \rightarrow^{S} \alpha_1 \rightarrow^{S} \alpha_2$$

However, there is more than one way that we might choose to annotate the *definition* of *twice*.

We might expect that, just as we pass binding-times explicitly in annotated programs, we should pass types explicitly to annotated polymorphic functions. Annotating *twice* in this way would result in something like

$$\mathit{twice}\ \alpha_1\ \alpha_2\ \beta\ f\ x = f@^{\beta}([\alpha_2 \mapsto \alpha_1]\ (f@^{\beta}x))$$

But notice that we need a coercion, which we have written as $[\alpha_2 \mapsto \alpha_1]$, between two unknown types here! The compiled code for a generating extension will need to construct representations of types during specialisation, pass them as parameters, and interpret them in order to implement such coercions. Because types may be complex, this may be expensive, and in any case we prefer to avoid interpretation in generating extensions.

Therefore, we treat polymorphic functions differently. Rather than passing *types* as parameters, we pass the necessary *coercion functions*, one for each subtype constraint in the function's type. With this idea, the annotated version of *twice* becomes

$$\mathit{twice}\ \beta\ \xi\ f\ x = f@^{\beta}(\xi\ (f@^{\beta}x))$$

where ξ implements the coercion $\alpha_2 \leq \alpha_1$. At each call of *twice*, we can pass a specialised coercion function for the types which actually occur.

3 Binding-Time Analysis

Binding-time annotations are usually constructed automatically by a *binding-time analyser*. We specify our polymorphic binding-time analysis via a type system for annotated programs, which guarantees that operations annotated as static never depend on dynamic values. Given an unannotated program, the binding-time analyser finds well-typed annotations that make as many operations as possible static. This type-based approach builds on earlier work by Dussart, Henglein and Mossin [12, 7], which has been adopted for the Similix partial evaluator [2]. We favour a type-based approach because it is efficient, comprehensible, and extends naturally to handle polymorphism.

We shall specify the binding-time type system for the smallest interesting language, and then discuss how it is used to infer annotations.

3.1 The Binding-Time Type System

We consider an annotated λ-calculus with polymorphic **let** and one base type:

$$\begin{array}{ll} e[\mathit{Expression}] & ::= c \mid x \mid \mathbf{let}\ x = e\ \mathbf{in}\ e \mid \lambda x.e \mid e@^{b}\phi\ e \mid \lambda\beta.e \mid e\ b \mid \lambda\xi.e \mid e\ \phi \\ b[\mathit{Binding\text{-}time}] & ::= S \mid D \mid \beta \mid b \sqcup b \\ \phi[\mathit{Coercion}] & ::= \iota \mid \xi \mid \mathit{Int}^{bb} \mid \phi \rightarrow^{bb} \phi \end{array}$$

Here β is a binding-time variable, ξ is a coercion variable, x is a program variable, and c is a constant.

In this simple language, only function application need be annotated with a binding-time, and only function arguments need be coerced. Constants and λ-expressions are always static, and are coerced to be dynamic where necessary. **let**-expressions are always dynamic, but their bodies may even so be static since we use Bondorf's CPS specialisation [3], which moves the context of a **let** into its body, where it can be specialised. Applications to binding-times and coercions always take place during specialisation, and so need no annotation.

We have already seen integer coercions. A coercion $\phi_1 \rightarrow^{b_1 b_2} \phi_2$ coerces a function with binding-time b_1 to one with binding-time b_2, applying coercion ϕ_1 to the argument and ϕ_2 to the result. ι is the identity coercion.

Binding-time types and constraints take the form

$$\begin{array}{ll} \tau[\mathit{Monotype}] & ::= \alpha \mid \mathit{Int}^{b} \mid \tau \rightarrow^{b} \tau \\ c[\mathit{Constraint}] & ::= b \leq b \mid b \rhd \tau \mid \phi : \tau \leq \tau \end{array}$$

The complete set of binding-time type inference rules can be found in the appendix; here we focus on the rule for application:

$$\frac{\Gamma; C \vdash e_1 : (\tau_1 \rightarrow \tau_2)^{b} \quad \Gamma; C \vdash e_2 : \tau_3 \quad C \vdash \phi : \tau_3 \leq \tau_1 \quad C \vdash b \rhd \tau_1 \quad C \vdash b \rhd \tau_2}{\Gamma; C \vdash (e_1\ @^{b}\ \phi\ e_2) : \tau_2}$$

As usual in a binding-time type system, our judgements depend both on an environment Γ and a set of constraints C. Notice, however, that our subtype constraints include the coercion that maps one type to the other. Thus, from the constraint set C, we infer *which* coercion ϕ converts τ_3 to τ_1. Notice also that we include $\triangleright$-constraints to guarantee that the type of the function is well-formed. Finally, the annotation on the application is taken from the type of the function.

Our constraint inference rules, with judgements of the form $C \vdash c$, can be found in the appendix. They are mostly standard, with the exception that the rules for subtyping actually construct a coercion. For example, the rule for function types

$$\frac{C \vdash \phi_1{:}\tau_3 \leq \tau_1 \quad C \vdash \phi_2{:}\tau_2 \leq \tau_4 \quad C \vdash b_1 \leq b_2}{C \vdash \phi_1 \rightarrow^{b_1 b_2} \phi_2 : \tau_1 \rightarrow^{b_1} \tau_2 \leq \tau_3 \rightarrow^{b_2} \tau_4}$$

constructs a coercion on functions from coercions on the argument and result. Where possible, we use the identity coercion

$$C \vdash \iota : \tau \leq \tau$$

which can be removed altogether by a post-processor. We restrict the coercions in C to be distinct coercion variables; thus we can think of C as a kind of environment, binding coercion variables to their types.

As in the Hindley-Milner type system, **let**-bound variables may have *type schemes* rather than monotypes. Type-schemes take the form

$$\begin{array}{ll}
\gamma[\textit{Qualified type}] & ::= \tau \mid q \Rightarrow \gamma \\
q[\textit{Qualifier}] & ::= b \leq b \mid b \triangleright \tau \mid \tau \leq \tau \\
\pi[\textit{Polychronic type}] & ::= \gamma \mid \forall \beta.\pi \\
\sigma[\textit{Polymorphic type}] & ::= \pi \mid \forall \alpha.\sigma
\end{array}$$

We give a complete set of rules to introduce and eliminate type-schemes in the appendix; note that although our rule system is not syntax-directed, it is easy to transform it into a syntax-directed system because of the restriction on where type schemes may appear. Here we discuss only the rules which are not standard.

Notice that qualifiers are almost, but not exactly, the same as constraints. The difference is that sub-type qualifiers $\tau_1 \leq \tau_2$ do not mention a coercion. Looking at the rules for introducing and eliminating such a qualifier

$$\frac{\Gamma; C, \xi{:}\tau_1 \leq \tau_2 \vdash e : \gamma}{\Gamma; C \vdash \lambda\xi.e : \tau_1 \leq \tau_2 \Rightarrow \gamma} \qquad \frac{\Gamma; C \vdash e : \tau_1 \leq \tau_2 \Rightarrow \gamma \quad C \vdash \phi{:}\tau_1 \leq \tau_2}{\Gamma; C \vdash e\ \phi : \gamma}$$

we see why: the coercion in the constraint becomes the bound variable of a coercion abstraction; it would be unnatural to allow bound variable names in types. That we 'forget' the coercion doesn't matter: it can be recreated where it is needed by the elimination rule.

The rules for generalising and instantiating type variables are standard, except that we only allow instantiation with well-formed types. The rules for binding-time variables just introduce binding-time abstraction and application:

$$\frac{\Gamma; C \vdash e : \gamma}{\Gamma; C \vdash \lambda\beta.e : \forall\beta.\gamma}\ \beta \notin FV(C, \Gamma) \qquad \frac{\Gamma; C \vdash e : \forall\beta.\gamma}{\Gamma; C \vdash e\ b : \gamma[b/\beta]}$$

Given any unannotated expression which is well-typed in the Hindley-Milner system, we can construct a well-typed annotated expression by annotating each application with a fresh binding-time variable and a fresh coercion variable, moving constraints into qualified types, and generalising all possible variables. But this leads to polymorphic definitions with very many generalised variables, and very many qualifiers. In the remainder of this section we will see how to reduce this multitude.

3.2 Simplifying Constraints

Before generalising the type of a **let**-bound variable, it is natural to simplify the constraints as much as possible. Simplification of this kind of constraint is mostly standard [11], except that we keep track of coercions also; essentially we use the constraint inference rules in the appendix backwards, instantiating variables where necessary to make rules match. For example, we simplify the constraint $\xi : \alpha \leq \mathit{Int}^b$ by instantiating α to Int^β and ξ to $\mathit{Int}^{\beta b}$, where β is fresh, and then simplifying the constraint to $\beta \leq b$. Simplification of this kind does not change the set of solutions of the constraints.

We use two non-standard simplification rules also. Firstly, whenever we discover a cycle of binding-time variables $\beta_1 \leq \cdots \leq \beta_n \leq \beta_1$, we instantiate each β_i to the same variable. We treat cycles of type variables similarly, which much reduces the number of variables we need to quantify over. Secondly, we simplify the constraints $\{D \triangleright \alpha_1, \xi : \alpha_1 \leq \alpha_2\}$ by instantiating α_2 to α_1 and ξ to ι: this preserves the set of solutions because both α_1 and α_2 have to be well-formed types annotated D at the top-level, and one such type can be a subtype of another only if they are equal.

Simplification terminates, which can be shown by a lexicographic argument: each rule reduces the size of types, the number of $\rhd$-constraints, the number of $\sqcup$s to the left of $\leq$, or the total number of constraints.

3.3 Simplifying Polymorphic Types

The simplifications in the previous section preserve the set of instances of a polymorphic type. That is, if we simplify a type scheme σ_1 to a type scheme σ_2, then any instance τ_1 of σ_1 is guaranteed also to be an instance of σ_2. But we can go further, if we guarantee only that there is an instance τ_2 of σ_2 which is a *subtype* of τ_1. This still enables us to use a polymorphic value of type σ_2 at any instance of σ_1, provided we introduce a coercion. For example, we can simplify the type of the *power* function from $\forall\beta_1,\beta_2,\beta_3.(\beta_1 \leq \beta_3, \beta_2 \leq \beta_3) \Rightarrow \mathit{Int}^{\beta_1} \rightarrow^S \mathit{Int}^{\beta_2} \rightarrow^S \mathit{Int}^{\beta_3}$ to $\forall\beta_1,\beta_2.\mathit{Int}^{\beta_1} \rightarrow^S \mathit{Int}^{\beta_2} \rightarrow^S \mathit{Int}^{\beta_1\sqcup\beta_2}$; these two types do not have the same instances, but any instance of the first can be derived by coercing an instance of the second. The second type has fewer quantified variables and coercions, and is therefore cheaper to specialise.

This subtype condition is guaranteed by ensuring that variables occurring negatively in the type are only instantiated to smaller quantities, while variables occurring positively are only instantiated to larger ones. Moreover, simplification must not increase the binding-time of any program annotation, otherwise it would lead to poorer specialisation. Positively occurring binding-time variables therefore cannot be instantiated at all. Dussart et al. [7] simplify by instantiating non-positive binding-time variables to the least upper bound of their lower bounds (as in the *power* example above).

In the presence of polymorphism, we instantiate type variables also. We might treat non-positive type variables in the same way that Dussart et al. treat binding-time variables, but this would introduce least upper bounds of type variables. This would be problematic for us, since we pass coercions and not types as parameters during specialisation: while it is straightforward (if expensive) to compute the least upper bound of two types, computing the least upper bound of two coercions would be far harder. But in two special cases, we can instantiate non-positive type variables to smaller types *without* needing least upper bounds.

Firstly, if $\xi : \alpha_1 \leq \alpha_2$ is the *only* constraint imposing a lower bound on α_2, and α_2 is non-positive, then we can instantiate α_2 to α_1 and ξ to ι. We also must insist that α_1 and α_2 are forced by the same set of binding-times; otherwise unifying them might make α_1 more dynamic.

Secondly, if α_1 and α_2 are *both* non-positive, have the same set of lower bounds, and are forced by the same binding-times, then they must take the same value in the least solution of the constraints, and we can unify them — even though we cannot express this least solution without least upper bound.

But there is another way to simplify constraints on type variables: we can instantiate non-negative type variables to *larger* types! This does potentially make some *types* more dynamic, but no *binding-times*, and it is the binding-time annotations which determine the quality of specialisation, not the types. We can do this in cases analogous to the two above, except that we need not be concerned with the binding-times which force type variables, since we *expect* to make type variables more dynamic. This process is specified formally in the appendix.

This form of simplification terminates since each step eliminates one variable.

For example, the type inferred for the *map* function, after simplifying binding-times, is

$$\begin{array}{l}\forall\alpha_1,\alpha_2,\alpha_3,\alpha_4,\alpha_5.\forall\beta_1,\beta_2.\\ \quad(\beta_1 \rhd \alpha_1, \beta_1 \rhd \alpha_2, \beta_2 \rhd \alpha_3, \beta_2 \rhd \alpha_4, \alpha_3 \leq \alpha_1, \alpha_2 \leq \alpha_4, \alpha_5 \leq \alpha_4) \Rightarrow\\ \quad\;(\alpha_1 \rightarrow^{\beta_1} \alpha_2) \rightarrow^S [\alpha_3]^{\beta_2} \rightarrow^S [\alpha_4]^{\beta_2}\end{array}$$

(where α_5 is a type variable internal to the definition of *map*). α_4 has two lower bounds, so cannot be reduced, while α_1 cannot be reduced to its only lower bound α_3 since $\beta_1 \rhd \alpha_1$, but β_1 does not force α_3. However, α_2, α_3, and α_5 are all non-positive and have unique upper bounds, so we can increase all three to their upper bounds and simplify the type to

$$\begin{array}{l}\forall\alpha_1,\alpha_2.\forall\beta_1,\beta_2.(\beta_1 \rhd \alpha_1, \beta_1 \rhd \alpha_2, \beta_2 \rhd \alpha_1, \beta_2 \rhd \alpha_2) \Rightarrow\\ \quad(\alpha_1 \rightarrow^{\beta_1} \alpha_2) \rightarrow^S [\alpha_1]^{\beta_2} \rightarrow^S [\alpha_2]^{\beta_2}\end{array}$$

The number of coercion parameters is decreased from three to zero.

4 Discussion

We have implemented this binding-time analysis in a prototype partial evaluator for polymorphic programs [10]. In practice, every binding-time analyser sometimes makes *too many* operations static, causing partial evaluation to loop, and ours is no exception. This must be prevented using user annotations, which have to be rethought in a polymorphic context. The full paper will contain details.

Polymorphism is particularly important for programs made up of many modules. In earlier work on specialising modules [8, 6, 9] we discovered we needed polymorphic binding-time analysis, which directly inspired this work.

Our analysis is built on Henglein et al's earlier polychronic analyses [12, 7]. Consel et al generalised their work in a different direction [4]. Binding-time analysers for polymorphic programs have also been developed based on abstract interpretation [15, 19], although this approach is now little used in practice.

This paper considers only parametric polymorphism, without overloading. We hope to extend our system to handle overloading based on Haskell classes [21].

Polymorphic typing is integral to widely used functional programming languages such as ML and Haskell, and has also been adopted in other languages such as Mercury [20]. Polymorphic binding-time analysis, such as ours, is vital if program specialisation is to be applied to such languages in practice.

References

1. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).
2. A. Bondorf. Automatic autoprojection of higher-order recursion equations. In N. Jones, editor, *3rd European Symposium on Programming*, LNCS, Copenhagen, 1990. Springer-Verlag.
3. A. Bondorf. Improving binding times without explicit cps-conversion. In *1992 ACM Conference on Lisp and Functional Programming. San Francisco, California*, pages 1–10, June 1992.
4. C. Consel and P. Jouvelot. Separate Polyvariant Binding-Time Analysis. Technical Report CS/E 93-006, Oregon Graduate Institute Tech, 1993.
5. Charles Consel. A tour of schism: a partial evaluation system for higher-order applicative languages. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154, June 1993.
6. D. Dussart, R. Heldal, and J. Hughes. Module-Sensitive Program Specialisation. In *Conference on Programming Language Design and Implementation*, Las Vegas, June 1997. ACM SIGPLAN.
7. D. Dussart, F. Henglein, and C. Mossin. Polymorphic Recursion and Subtype Qualifications: Polymorphic Binding-Time Analysis in Polynomial Time. In Alan Mycroft, editor, *SAS'95: 2nd Int'l Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 118–135, Glasgow, Scotland, September 1995. Springer-Verlag.
8. R. Heldal and J. Hughes. Partial Evaluation and Separate Compilation. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Amsterdam, June 1997. ACM SIGPLAN.
9. R. Heldal and J. Hughes. Extending a partial evaluator which supports separate compilation. *Theoretical Computer Science 248*, pages 99–145, 2000.
10. Rogardt Heldal. *The Treatment of Polymorphism and Modules in a Partial Evaluator.* PhD thesis, Chalmers University of Technology, April 2001.
11. F. Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *FPCA*, pages 448–472. 5th ACM Conference, Cambridge, MA, USA, Springer-Verlag, August 1991. Lecture Notes in Computer Science, Vol. 523.
12. F. Henglein and C. Mossin. Polymorphic Binding-Time Analysis. In D. Sannella, editor, *ESOP'94: European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 287–301. Springer-Verlag, April 1994.
13. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.
14. Simon Peyton Jones, John Hughes, (editors), Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language. Available from `http://haskell.org`, February 1999.
15. J. Launchbury. *Projection Factorisations in Partial Evaluation (PhD thesis)*, volume 1 of *Distinguished Dissertations in Computer Science.* Cambridge University Press, 1991.
16. K. Malmkjr, N. Heintze, and O. Danvy. ML partial evaluation using set-based analysis. In *Workshop on ML and its Applications*, pages 112–119. ACM SIGPLAN, 1994.
17. R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and Systems Sciences*, 17:348–375, 1978.

18. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.
19. T. Æ. Mogensen. Binding Time Analysis for Polymorphically Typed Higher Order Languages. In *Theory and Practice of Software Development*, volume 352 of *Lecture Notes in Computer Science*, pages 298–312. Springer-Verlag, March 1989.
20. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, pages 499–512, Glenelg, Australia, February 1995.
21. P. Wadler and S. Blott. How to make *ad hoc* polymorphism less *ad hoc.* In *Proceedings 1989 Symposium Principles of Programming Languages*, pages 60–76, Austin, Texas, 1989.

A Appendix: Binding-Time Rules

$$\Gamma, x{:}\sigma, \Gamma'; C \vdash x : \sigma \qquad \Gamma; C \vdash c : Int^S \qquad \frac{C \vdash \tau_1 \text{ wft} \quad \Gamma, x{:}\tau_1; C \vdash e : \tau_2}{\Gamma; C \vdash \lambda x.e : (\tau_1 \to^S \tau_2)}$$

$$\frac{\Gamma; C \vdash e_1 : (\tau_1 \to \tau_2)^b \quad \Gamma; C \vdash e_2 : \tau_3 \quad C \vdash \phi : \tau_3 \leq \tau_1 \quad C \vdash b \rhd \tau_1 \quad C \vdash b \rhd \tau_2}{\Gamma; C \vdash (e_1 \mathbin{@^b} \phi\ e_2) : \tau_2}$$

$$\frac{\Gamma; C \vdash e_1 : \sigma \quad \Gamma, x{:}\sigma; C \vdash e_2 : \tau}{\Gamma; C \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau}$$

Fig. 1. Syntax Directed Binding-time Rules for Expressions.

$$\frac{\Gamma; C \vdash e : \gamma}{\Gamma; C \vdash \lambda\beta.e : \forall\beta.\gamma}\ \beta \notin FV(C, \Gamma) \qquad \frac{\Gamma; C \vdash e : \forall\beta.\gamma}{\Gamma; C \vdash e\ b : \gamma[b/\beta]}$$

$$\frac{\Gamma; C, b_1 \leq b_2 \vdash e : \gamma}{\Gamma; C \vdash e : b_1 \leq b_2 \Rightarrow \gamma} \qquad \frac{\Gamma; C, b \rhd \tau \vdash e : \gamma}{\Gamma; C \vdash e : b \rhd \tau \Rightarrow \gamma} \qquad \frac{\Gamma; C, \xi{:}\tau_1 \leq \tau_2 \vdash e : \gamma}{\Gamma; C \vdash \lambda\xi.e : \tau_1 \leq \tau_2 \Rightarrow \gamma}$$

$$\frac{\Gamma; C \vdash e : b_1 \leq b_2 \Rightarrow \gamma \quad C \vdash b_1 \leq b_2}{\Gamma; C \vdash e : \gamma} \qquad \frac{\Gamma; C \vdash e : b \rhd \tau \Rightarrow \gamma \quad C \vdash b \rhd \tau}{\Gamma; C \vdash e : \gamma}$$

$$\frac{\Gamma; C \vdash e : \tau_1 \leq \tau_2 \Rightarrow \gamma \quad C \vdash \phi{:}\tau_1 \leq \tau_2}{\Gamma; C \vdash e\ \phi : \gamma}$$

$$\frac{\Gamma; C \vdash e : \sigma}{\Gamma; C \vdash e : \forall\alpha.\sigma}\ \alpha \notin FV(C, \Gamma) \qquad \frac{\Gamma; C \vdash e : \forall\alpha.\sigma \quad C \vdash \tau \text{ wft}}{\Gamma; C \vdash e : \sigma[\tau/\alpha]}$$

Fig. 2. Non-Syntax Directed Rules

$$C, c \vdash c \qquad C \vdash \iota : \tau \leq \tau \qquad \frac{C \vdash b_1 \leq b_2}{C \vdash Int^{b_1 b_2} : Int^{b_1} \leq Int^{b_2}}$$

$$\frac{C \vdash \phi_1{:}\tau_3 \leq \tau_1 \quad C \vdash \phi_2{:}\tau_2 \leq \tau_4 \quad C \vdash b_1 \leq b_2}{C \vdash \phi_1 \to^{b_1 b_2} \phi_2 : \tau_1 \to^{b_1} \tau_2 \leq \tau_3 \to^{b_2} \tau_4}$$

$$\frac{C \vdash b_1 \leq b_2}{C \vdash b_1 \rhd Int^{b_2}} \qquad C \vdash S \rhd \tau \qquad \frac{C \vdash b_1 \leq b_2}{C \vdash b_1 \rhd \tau_1 \to^{b_2} \tau_2}$$

$$C \vdash b \leq b \qquad C \vdash S \leq b \qquad C \vdash b \leq D \qquad C \vdash \beta_i \leq \sqcup\beta_i \qquad \frac{C \vdash \beta_1 \leq \beta_3 \quad C \vdash \beta_2 \leq \beta_3}{C \vdash \beta_1 \sqcup \beta_2 \leq \beta_3}$$

Fig. 3. Constraint Inference Rules

$$C \vdash \alpha \text{ wft} \quad C \vdash \mathit{Base}^b \text{ wft} \quad \frac{C \vdash \tau_1 \text{ wft} \;\; C \vdash \tau_2 \text{ wft} \;\; C \vdash b \rhd \tau_1 \;\; C \vdash b \rhd \tau_2}{C \vdash \tau_1 \rightarrow^b \tau_2 \text{ wft}}$$

Fig. 4. Well-formedness of Types.

Each time one of the rules below is applied, the constraints must first be normalised and the set of force constraints must be closed using the following rule:

$$\{\beta \rhd \alpha_1, \xi : \alpha_1 \leq \alpha_2\} \rightsquigarrow \{\beta \rhd \alpha_1, \beta \rhd \alpha_2, \xi : \alpha_1 \leq \alpha_2\}$$

To simplify a type τ and constraint set C in an environment Γ:

$$\beta \notin (|\tau|^- \cup FV(\Gamma)) \Rightarrow C \rightsquigarrow C_\beta[\beta := \sqcup_{\beta' \in C_{\leq\beta}} \beta']; \beta := \sqcup_{\beta' \in C_{\leq\beta}} \beta'$$

$$\begin{array}{l} \alpha \notin (|\tau|^- \cup FV(\Gamma)) \;\wedge\; C_{\leq\alpha} = \{\} \;\wedge\; C_{\rhd\alpha} \subseteq C_{\rhd\alpha_1} \\ \Rightarrow C, \xi : \alpha_1 \leq \alpha \rightsquigarrow C[\alpha := \alpha_1]; \alpha := \alpha_1, \xi := \iota \end{array}$$

$$\begin{array}{l} \alpha_1, \alpha_2 \notin (|\tau|^- \cup FV(\Gamma)) \;\wedge\; C_{\leq\alpha_1} = C_{\leq\alpha_2} \;\wedge\; C_{\rhd\alpha_1} = C_{\rhd\alpha_2} \\ \Rightarrow C \rightsquigarrow C[\alpha_1 := \alpha_2]; \alpha_1 := \alpha_2 \end{array}$$

$$\begin{array}{l} \alpha \notin (|\tau|^+ \cup FV(\Gamma)) \wedge\; C_{\alpha\leq} = \{\} \\ \Rightarrow C, \xi : \alpha \leq \alpha_1 \rightsquigarrow C[\alpha := \alpha_1]; \alpha := \alpha_1, \xi := \iota \end{array}$$

$$\begin{array}{l} \alpha_1, \alpha_2 \notin (|\tau|^+ \cup FV(\Gamma)) \;\wedge\; C_{\alpha_1\leq} = C_{\alpha_2\leq} \\ \Rightarrow C \rightsquigarrow C[\alpha_1 := \alpha_2]; \alpha_1 := \alpha_2 \end{array}$$

where

$$\begin{array}{lll} C_{\leq\beta} \triangleq \{\beta_1 | \beta_1 \leq \beta \in C\} & C_\beta \triangleq C - \{\beta_1 \leq \beta \,|\beta_1 \in \mathbb{B}\} & \\ C_{\leq\alpha} \triangleq \{\alpha_1 | \xi : \alpha_1 \leq \alpha \in C\} & C_{\rhd\alpha} \triangleq \{b | b \rhd \alpha \in C\} & C_{\alpha\leq} \triangleq \{\alpha_1 | \xi : \alpha \leq \alpha_1 \in C\} \end{array}$$

Fig. 5. Increasing and Decreasing Variables.

An Investigation of Compact and Efficient Number Representations in the Pure Lambda Calculus

Torben Æ. Mogensen

DIKU, University of Copenhagen, Denmark
Universitetsparken 1, DK-2100 Copenhagen O, Denmark
phone: (+45) 35321404, fax: (+45) 35321401, e-mail: torbenm@diku.dk

Abstract. We argue that a right-associated binary number representation gives simpler operators and better efficiency than the left-associated binary number representation investigated by Goldberg. We generalise this representation to higher number-bases and show that bases between 3 and 5 can give higher efficiency than binary representation.

1 Introduction

The archetypal number representation in the pure lamda calculus is Church numerals. The definitions of addition, multiplication and raising to power are extremely simple when using Church numerals, but numbers are not represented very compactly and the operations, though simple, are quite costly.

den Hoed [4] suggested a compact representation of binary numbers in the pure untyped lambda calculus, where the binary number $b_n \dots b_1 b_0$ is represented as $\lambda x_0 x_1.x_{b_0}\ x_{b_1} \dots x_{b_n}$. Note that, due to the convention that application is left-associative, the term is read as $\lambda x_0 x_1.((x_{b_0}\ x_{b_1}) \dots x_{b_n})$. Hence, this representation is called *left-associated.* Mayer Goldberg [1] has shown that efficient operators exist for this representation.

We believe we can achieve better by using a right-associated representation. Furthermore, we introduce one more variable to mark the end of a bit sequence: 0 is represented by $\lambda z x_0 x_1.z$ and $b_n \dots b_1 b_0$, where $b_n \neq 0$ is represented by $\lambda z x_0 x_1.x_{b_0}\ (x_{b_1}\ (\dots (x_{b_n}\ z))$.

We use standard lambda calculus notation. We use booleans: $T \equiv \lambda xy.x$, $F \equiv \lambda xy.y$, pairs: $[e_1, e_2] \equiv \lambda x.x\ e_1\ e_2$ and projections: $\pi^2_k \equiv \lambda t.t(\lambda x_1 x_n.x_k)$. The identity function is $I \equiv \lambda x.x$. We use the notation $\lceil n \rceil$ to mean the representation of the number n. Example: $\lceil 5 \rceil \equiv \lambda z x_0 x_1.x_1\ (x_0\ (x_1\ z))$

2 Basic Operations on Numbers

Shifting a binary number up by one bit is easily done in constant time:

$$\uparrow \equiv \lambda bn.\lambda z x_0 x_1.b\ x_0\ x_1\ (n\ z\ x_0\ x_1)$$

Single bits are represented as $0 \equiv T$, $1 \equiv F$. For brevity and slightly better efficiency, we will often use specialised versions of $\uparrow$: $\uparrow_0 = \uparrow 0$ and $\uparrow_1 = \uparrow 1$.

For reasons of space, we will only show the operators that show that the representation is an adequate number system: zero-testing, increment and decrement. The operators apply a number to three terms: One for handling the empty bit string, one for handling the case where the least significant bit is 0 and one for the case where it is 1. In the *succ* operator we build a pair containing n and $n+1$ and select the one we need at each step. *pred*, similarly builds a pair of n and $n-1$. *pred* can introduce a leading zero if the number is a power of two.

$$zero? \equiv \lambda n.n\ T\ I\ (\lambda x.F)$$

$$\begin{array}{llll}
succ \equiv \lambda n.\pi^2_2\ (n\ Z\ A\ B) & \text{where} & pred \equiv \lambda n.\pi^2_2\ (n\ Z\ A\ B) & \text{where} \\
Z \quad \equiv [\lceil 0 \rceil, \lceil 1 \rceil] & & Z \quad \equiv [\lceil 0 \rceil, \lceil 0 \rceil] & \\
A \quad \equiv \lambda p.p\ (\lambda nm.[\uparrow_0 n, \uparrow_1 n]) & & A \quad \equiv \lambda p.p\ (\lambda nm.[\uparrow_0 n, \uparrow_1 m]) & \\
B \quad \equiv \lambda p.p\ (\lambda nm.[\uparrow_1 n, \uparrow_0 m]) & & B \quad \equiv \lambda p.p\ (\lambda nm.[\uparrow_1 n, \uparrow_0 n]) &
\end{array}$$

3 Other Number Bases

If we have an n-digit number $d_{n-1} \ldots d_1 d_0$ in base b, we can represent this as

$$\lambda z x_0 x_1 \ldots x_{n-1}.x_{d_0}\ (x_{d_1}\ (\ldots (x_{d_{n-1}}\ z) \ldots)).$$

0 is represented as $\lambda z x_0 x_1 \ldots x_{n-1}.z$. The operations on binary numbers are easily generalised to higher bases, but are not shown due to space restrictions.

Each digit in a base b number is represented by a variable and an application. This could lead us to believe that we can get arbitrary compactness by choosing higher bases. But an actual representation of a lambda term on a machine will have to represent variables using a fixed alphabet, so the number of symbols needed to represent a variable depend on the number of different variables. In order to be precise about this, we will use de Bruijn notation: each occurrence of a variable is replaced by a (de Bruijn) number that counts the number of lambda abstractions one passes in the syntax tree up to the abstraction that binds the variable (which is no longer named there). We have, so far, just replaced an unbounded number of variables by an unbounded number of de Bruijn numbers. We can, however, replace these by number strings. We will use unary representation for de Bruijn numbers, such that, *e.g.*, 3 is represented as *sssz*. While normal λ-notation omits left-associated brackets while right-associated brackets need to be made explicit, we will make each application explicitly bracketed. By making the closing bracket explicit and omitting the opening bracket, we get reverse-polish notation. Examples:

lambda term	de Bruijn notation	compact representation
$\lambda x.x$	λ0	λz
$\lambda z x_0 x_1.x_1\ (x_0\ (x_1\ z))$	λλλ0 (1 (0 2))	$\lambda\lambda\lambda zszzssz)))$

An n-bit number is represented by a string of length $2.5 \times n + 1$ on average for den Hoed's left-associated representation. The right-associated representation requires $2.5 \times n + 6$ symbols on average for the same. Given that a base-b digit and an application takes on average $(b{+}3)/2$ symbols to write and that you need $n{*}ln(2)/ln(b)$ base-b digits to represent an n-bit number, we get the following measure of compactness for base-b representation:

base	2	3	4	5	6	7	8	9	10
compactness	2.5	1.89	1.75	1.72	1.74	1.78	1.83	1.89	1.96

We get the asymptotically most compact representation if we use base 5 representation. If the of processing a digit is $kb+l$, the cost of processing an n-bit number (in base b) is $(kb+l)ln(2)/ln(b)$. The minimum is obtained when $b(ln(b)-1) = l/k$. The minima for a number of different l/k are:

l/k	0	1/4	1/2	2/3	1	3/2	2	3	4
minimum	2.72	2.95	3.18	3.32	3.59	3.97	4.32	4.97	5.57
optimal base	3	3	3	3	4	4	4	5	6

The minimum is always at b higher than 2. This indicates that it will always be better to use base 3 or higher instead of base 2. Note that these measures are about asymptotic costs. For small numbers it will be better to use a small number base, such as binary or even unary.

Balanced ternary is a variant base-3 notation which has digits "-" (representing -1), "0" and "+" (representing 1). This simplifies negative numbers, as no separate sign symbol is required. Balanced ternary has been used in some early Russian computers and may well be a good number representation for the lamda calculus. It isn't hard to modify our representation scheme and operators to balanced ternary.

4 Binary Operators

Some binary operators, like equality, addition and subtraction, require walking down two digit strings simultaneously. The representations we have shown so far aren't geared to this, so we introduce yet another representation (not shown for space reasons), which allows us to inspect one digit at a time. We can define operators which takes a number n in the "normal" representation ($\lceil n \rceil$) to the new representation or *vice-versa*. The idea is that one of the arguments to a binary operator is converted to the new representation while the other is processed directly in the original representation. We then use this to define an operator that takes both arguments in the original representation. Multiplication is simple to define once we have addition and subtraction. For reasons of space, we have omitted the definitions of these operators.

5 Benchmarks

We have timed some calculations using different representations. To execute the calculations, we have used a lambda normaliser based on normalisation by evaluation and implemented in scheme [3]. The test we use is counting from 0 to 50000 using the *succ* operator.

Base	2	3	4	5	6	balanced 3
Time	6270 ms	4380 ms	4000 ms	3900 ms	4470 ms	4660 ms

The time used to execute the benchmark drops by more than 30% from binary to base 3, but the advantage of further going to base 4 or 5 is less (around 10%).

This supports the conjecture that the optimal base is higher than 2 and likely to be around 4 or 5. Balanced ternary is slightly slower than ordinary ternary, but that should be no surprise since the benchmark doesn't use negative numbers.

6 Conclusion

We have investigated a number of different compact number representations for the lambda calculus, starting with the left-associated binary number system suggested by den Hoed. We argued that we get better calculation efficiency by choosing a right-associated representation and adding an explicit end symbol. We then found that number bases in the range 3-6 increase compactness and calculation efficiency over binary representation.

While execution efficiency seems optimal at base 4 or 5, the operators become much bigger in these bases than in ternary: The size of the *succ* operator is approximately quadratic in the number base, and the size of the addition operator is approximately cubic in the number base. This may make base 3 the overall best choice. If ease of conversion to/from binary notation is important, a base-4 representation might be preferable.

The ease of handling negative numbers leads us to suggest using a balanced ternary number representation, for which we present some binary operators in addition to the unary operators we presented for the other systems.

While we, arguably, gain efficiency over Goldbergs operators for den Hoed's representation, this may not be an entirely fair comparison: After all, Goldbergs work was an answer to a challenge if he could make decent operations for a specific number system that was not designed for that purpose. Hence, he didn't *a priori* have the freedom we have exploited of changing the number system to gain better efficiency.

References

1. Mayer Goldberg. An adequate and efficient left-associated binary numreal system in the λ-calculus. *Journal of Functional Programming*, 10(6):607–623, November 2000.
2. T. Æ. Mogensen. Self-applicable online partial evaluation of the pure lambda calculus. In William L. Scherlis, editor, *Proceedings of PEPM '95*, pages 39–44. ACM, ACM Press, 1995.
3. T. Æ. Mogensen. G¡delisation in the untyped lambda calculu. In O. Danvy, editor, *Proceedings of PEPM'99*. BRICS Notes Series, 1999.
4. W. L. van den Poel, C. E. Schaap, and G. van der Mey. New arithmetical operators in the theory of combinators. *Indagationes Mathematicae*, (42):271–325, 1980.

Processes and Concurrency

Communicating Generalised Substitution Language

Andy Galloway

High Integrity Systems Engineering,
Department of Computer Science,
University of York, UK
fax: +44 1904 432708, e-mail: andyg@cs.york.ac.uk.

1 Introduction

> "Many of the older programming languages provide jumps to labels in addition to more abstract program structures. Such a mixture of levels is not always recommended in programming practice, but it provides a good exercise for the extensibility of our theory" (C.A.R Hoare and He J., Unifying Theories of Programming [HJ98])

We present an integration of the process algebra value-passing CCS [Mil89] with the model-based method B's Generalised Substitution Language (GSL) [Abr96]. We acknowledge that all value passing CCS specifications are semantically equivalent to non-deterministic sequential program specifications with goto control structures (with input/output not constrained to the beginning and end of the program). Our approach will be to use a symbolic operational semantics (as inspired by Hennessy, Lin and Rathke [HL95,Lin96,RH96]) to define the sequential program to which an arbitrarily distributed system corresponds (represented by a symbolic labelled transition system), and then use the generalised substitution (weakest precondition) calculus to reason about the behaviour of the sequential program.

The approach has been developed as part of a collaborative project between the UK Ministry of Defence, Rolls-Royce plc and BAE SYSTEMS. The project aims to provide *practical* formal approaches to the validation and verification of distributed control systems, such as aircraft engine control systems and on-board aircraft systems. This presentation will concentrate on an untimed model of computation, presenting some simple healthiness conditions (deadlock and divergence freedom). Future papers will deal with refinement proof obligations, how time is used to augment the model, and how the theory is applied to the domain. In the full paper, we will also present a comparison with the Abstract Systems (Action Systems) B approach.

We choose to follow an operational rather than denotational semantic style despite the limitations of the approach with respect to proving properties which require induction. There are two reasons for this. Firstly, the class of problems we have encountered so far in the domain are ones which may be specified without recursion over the static operators of the process algebra, therefore finite symbolic labelled transition systems may always be constructed. Secondly, in accordance with our remit to be *practical* in our proposed methods, we are trying to reduce the complexity of the required mathematical proof as much as possible by automatic means. Model-checking and decision procedures aid this process. By using a symbolic operational approach, we aim to distill automatically the aspects of the proof that concern concurrency and communication leaving the raw data relationships which must hold as proof obligations. Such a separation also means that we can use model-checking both at the process algebra level (e.g. ground semantics, data independence) and at the set-theory proof level (heuristically aided model search).

2 The Syntax of Communicating Generalised Substitution Language

We describe the syntax of Communication Generalised Substitution Language (CGSL) as a dialect of value-passing CCS. The main difference is the inclusion of a preconditioned agent and a substitution prefix. The preconditioned agent describes an agent which diverges if its precondition is not true (we can describe this behaviour without the precondition syntax, but it provides a convenient shorthand which is mirrored in GSL). Conversely, the guarded agent deadlocks if its guard is not true. The substitution prefix allows a GSL operation to be associated with a τ action, updating the variables it operates on according to its specification before evolving into the agent it prefixes.

The syntax of CGSL is built upon a given set of Action names (*ActionName*) and Agent constants (*AgentName*) and the Generalised Substitution Language syntax (*GSL*, as presented in [Abr96]). We also use the sub-syntactic categories *Id_List*, a (possibly empty) list of variable names, *Predicate* and *Expression* from GSL.

The syntax is given as:

Agent ::=	0	Deadlock
	$ActionName(Id\_List \mid Predicate).Agent$	Input Prefix
	$\overline{ActionName}\ (Expression).Agent$	Output Prefix
	$\tau.Agent$	Silent action Prefix
	$[GSL].Agent$	Substitution Prefix
	$Agent + Agent$	Choice
	$Predicate \Longrightarrow Agent$	Guarded Agent
	PRE *Predicate* **THEN** *Agent*	Preconditioned Agent
	$Agent \mid Agent$	Parallel Composition
	$Agent \backslash ActionSet$	Restriction
	$AgentName(Expression)$	Agent Constant

$$
\begin{aligned}
&ActionSet \;\; ::= \{ActionName_1 \langle, .., ActionName_n \rangle\} \\
&AgentDef \;\; ::= AgentName(Id\_List \mid Predicate) \overset{\text{def}}{=} Agent \\
&AgentsSpec ::= AgentDef_1 \langle, .., AgentDef_n \rangle
\end{aligned}
$$

Note that we do not include a syntactic construction for basic actions and agents (with no value's associated). We model such actions and agents with the empty variable list (for input actions and agent declarations), and the 'empty expression' (for output actions and agent references). We leave what we mean by the empty expression somewhat vague for now. Our choice depends upon what kind of expression we choose to complement a declaration. Ideally, (and our choice when we implement the system), will be to use the schema binding construct from Z [Spi95]. However, unlike Z [Spi95], B does not have the schema binding construct in its underlying set theory, otherwise we could use the empty schema binding to correspond to the 'empty expression' and the empty schema to correspond to the empty declaration. Alternatively, we could use tuples as expressions, in line with B's set theoretic model (pre and rel). The empty declaration would be represented by an empty variable list, but we would have to admit the empty tuple '()' in our underlying set theory. The choice of complement expression also affects any type-checking regime we wish to place on agent definitions.

For this presentation we leave the choice open. We assume that for a declaration containing the variable list $x_1, \ldots x_n$ there is enough information in the complement expression type (either a tuple $(e_1, \ldots, e_n)$ or a binding $\langle\!| \, x_1 == e_1, \ldots, x_n == e_n \, |\!\rangle$) to construct a multiple substitution in the GSL style $x_1, \ldots, x_n := e_1, \ldots, e_n$ (with *skip* corresponding to the substitution on the empty variable list). We also leave the type-checking details to future publications.

3 The Computational Model

The computational model is different from that of standard value-passing CCS, in which common occurrences of variable names on different sides of the parallel operator refer to different variables. In CGSL common variable names always refer to the same variable regardless of their position in an agent containing parallel operators. This means that the computational model also admits specifications of computations on shared variable spaces. For instance, the agent:

$$[x := 1].0 \mid [x := 2].\ \overline{out}\ (x).0$$

is capable of producing an $\overline{out}$ (1) as well as an $\overline{out}$ (2) depending upon how it interleaves. In general, the computational model is of a common global variable space which is referenced and updated by the behaviour of any of the parallel processes. Despite this, sensible (and checkable) variable naming (in the case of agents which are non-recursive over static operators) can ensure that parts of that space can only be referenced and altered by particular (sets of) processes. Indeed, alpha-conversion preprocessing of variables (on the non-static-recursive agents) can be used to produce the standard value-passing CCS interpretation for a CGSL specification.

The fact that variables are shared is natural. The labelled transition system defined by the symbolic operational semantics is an abstract representation of a non-deterministic sequential program with goto control structures. The arcs of the transition system correspond to program (specification) statements and input/output actions. The termination condition (trm(S) in B) of the specification statements associated with each arc define the conditions under which the system will diverge. The feasibility of the specification (fis(S) in B) associated with each arc defines when the transition is enabled. The body of each specification represents a (possibly non-deterministic) assignment of values to variables. The symbolic operational semantics defines communication synchronisation, in the corresponding sequential system, as assignment of an expression made up from one part of the variable space to a variable in another part of the variable space. It is natural, given this, to allow direct assignment of shared variables and interpret these too as simple assignments in the corresponding sequential system[1].

4 The Semantics of CGSL

The symbolic operational semantics defines a labelled transition system:

$$\mathcal{L} \in (Agent \times GSL \times Action) \leftrightarrow Agent$$

Where *Action* is the set $(ActionName \times (Id\_List \times Predicate)) \cup (\overline{ActionName} \times Expression) \cup \{\tau\}$

We write:

$$E \xrightarrow{[S],a} E'$$

for $((E, S, a), E') \in \mathcal{L}$.

For a given agent specification (*Spec* : *AgentSpec*) we define $\mathcal{L}$ as the smallest LTS which satisfies the operational rules below, containing all of the agent constants defined by *Spec* and containing any agents reachable from those constants. We write X := e for the multiple substitution of the variables in X (*Id_List*) with the values in the complement expression e, and $e \in \{X \mid P\}$ to mean the complement expression e satisfies the predicate constraint P. For the restriction rule we write: $Actionof(a)$ for the *ActionName* associated with action a, and $ActionSetof(M)$ for the set of all action names (and their compliments) mentioned in M.

The rules are shown in table 1.

5 Some Simple Validation Conditions

5.1 Symbolic Executions

Definition A symbolic execution of a transition system $\mathcal{L}$ is a total function from the nodes of the transition system[2] to the set of predicates (as operated on by the generalised substitution language).

$$SymbolicExecution(\mathcal{L}) == \mathcal{L}^{\dagger} \rightarrow Predicate$$

Symbolic executions are constructed to prove that particular properties are invariant across the entire transition system, such as deadlock and divergence freedom. In each symbolic execution, the predicate associated with each node of the transition system should be stronger than each of the weakest preconditions its exit transitions need to establish the predicate associated with their target node.

[1] Note that the interleaving semantics of CCS guarantees mutual exclusion of writes to variables except in one case, the synchronisation case. Mutual exclusion is guaranteed by the side condition associated with the B parallel substitution that the parallel substitutions must operate on distinct variables. Synchronisation transitions which break the mutual exclusion property are disallowed.

[2] We define $\mathcal{L}^{\dagger}$ as $\{E \mid \exists E', S, a \bullet (E, S, a) \mapsto E' \in \mathcal{L}\} \cup \operatorname{ran} \mathcal{L}$

Table 1. The Symbolic Operational Semantics for CGSL

$$(1)\quad \frac{}{\overline{a}(e).E \xrightarrow{[skip], \overline{a}(e)} E}$$

Output action prefix

$$(2)\quad \frac{}{\tau.E \xrightarrow{[skip], \tau} E}$$

Silent Action prefix

$$(3)\quad \frac{}{a(X|P).E \xrightarrow{[Skip], a(X|P)} E}$$

Input action prefix

$$(4)\quad \frac{}{[S].E \xrightarrow{[S], \tau} E}$$

Substitution Prefix

$$(5)\quad \frac{E \xrightarrow{[S], a} E'}{E + F \xrightarrow{[S], a} E'}$$

Left Choice

$$(6)\quad \frac{E \xrightarrow{[S], a} E'}{F + E \xrightarrow{[S], a} E'}$$

Right Choice

$$(7)\quad \frac{E \xrightarrow{[S], a} E'}{P \Longrightarrow E \xrightarrow{[P \Longrightarrow S], a} E'}$$

Guarded Agent

$$(8)\quad \frac{E \xrightarrow{[S], a} E'}{\mathbf{PRE}\ P\ \mathbf{THEN}\ E \xrightarrow{[P|S], a} E'}$$

Preconditioned Agent

$$(9)\quad \frac{E \xrightarrow{[S], a} E'}{E|F \xrightarrow{[S], a} E'|F}$$

Parallel Left

$$(10)\quad \frac{E \xrightarrow{[S], a} E'}{F|E \xrightarrow{[S], a} F|E'}$$

Parallel Right

$$(11)\quad \frac{E \xrightarrow{[S], a(X|P)} E' \qquad F \xrightarrow{[T], \overline{a}(f)} F'}{E|F \xrightarrow{[Sub], \tau} E'|F'} \qquad \begin{array}{c} \scriptstyle Sub=[(S||T);\, f \in \{X|P\} \Longrightarrow X := f] \\ \textit{S and T must operate on distinct variables} \end{array}$$

Parallel Synch Left

$$(12)\quad \frac{E \xrightarrow{[S], a(X|P)} E' \qquad F \xrightarrow{[T], \overline{a}(f)} F'}{F|E \xrightarrow{[Sub], \tau} F'|E'} \qquad \begin{array}{c} \scriptstyle Sub=[(S||T);\, f \in \{X|P\} \Longrightarrow X := f] \\ \textit{S and T must operate on distinct variables} \end{array}$$

Parallel Synch Right

$$(13)\quad \frac{E \xrightarrow{[S], a} E'}{E \backslash M \xrightarrow{[S], a} E' \backslash M} \qquad Actionof(a) \notin ActionSetof(M)$$

Restriction

$$(14)\quad \frac{E \xrightarrow{[S], a} E'}{A(e) \xrightarrow{[T], a} E'} \qquad A(X \mid P) \overset{\text{def}}{=} E,\ T = [e \in \{X \mid P\} \Longrightarrow X := e;\ S]$$

Agent Definition (Action)

Non-Divergent Agents and Operations In fact, the weakest precondition property alone is sufficient to guarantee that operations and agents are always used within their preconditions (although it is not sufficient to show that agents do not demonstrate divergent behaviour).

We begin by formalising this property. An operation-healthy (OH) symbolic execution $\mathcal{E}$ with respect to the transition system $\mathcal{L}$:

$$\mathcal{E} : SymbolicExecution(\mathcal{L})$$

is one in which for all pairs of nodes E, E' in dom $\mathcal{E}$, input actions a, Id_Lists X, Predicates P, output actions $\overline{a}$ and expressions e:

$$E \xrightarrow{[S], a(X)} E' \Rightarrow (\mathcal{E}(E) \Rightarrow [@X \bullet X \in \{X \mid P\} \Longrightarrow S]\mathcal{E}(E'))$$

$$E \xrightarrow{[S], \overline{a}(e)} E' \Rightarrow (\mathcal{E}(E) \Rightarrow [S]\mathcal{E}(E'))$$

$$E \xrightarrow{[S], \tau} E' \Rightarrow (\mathcal{E}(E) \Rightarrow [S]\mathcal{E}(E'))$$

We take the liberty of overloading the rightmost implication operator and substitution operator $[S]$ as a relation and function on the set *Predicate* rather than as a predicate and predicate transformer respectively. Pragmatically, the set relation and operator definitions tell the user how predicate and predicate transformers are to be used when proving the healthiness of a symbolic execution.

An operation-healthy symbolic execution guarantees the precondition of every operation invoked (and every Agent visited) whatever order the transition system allows them to be invoked.

Note also that GSL's generalised choice operator '@' is used to non-deterministically choose a value when the action is an input action from the environment.

Deadlock Freedom We extend the notation of an operation-healthy symbolic execution to a "deadlock free operation-healthy (DFOH) symbolic execution" by adding constraints that ensure there is always at least one transition from every node. This is achieved by showing that there is at least one transition where it is *possible* to establish the predicate *True*: An deadlock free operation-healthy symbolic execution $\mathcal{D}$ with respect to the transition system $\mathcal{L}$:

$$\mathcal{D} : \mathit{SymbolicExecution}(\mathcal{L})$$

is one which is operation-healthy and in addition for which for all nodes E:

$$\begin{aligned}\mathcal{D}(E) \Rightarrow (& \bigvee_{E', S, a, X \mid E \xrightarrow{[S], a(X)} E'} \neg\, [@\mathit{vars}(X) \bullet \mathit{vars}(X) \in T(X) \Longrightarrow S]\mathit{False} \vee \\ & \bigvee_{E', S, \overline{a}, e \mid E \xrightarrow{[S], \overline{a}(e)} E'} \neg\, [S]\mathit{False} \vee \\ & \bigvee_{E', S \mid E \xrightarrow{[S], \tau} E'} \neg\, [S]\mathit{False})\end{aligned}$$

The above uses the double negation property of the generalised substitution language. Whilst $[S]P$ yields the weakest property needed to satisfy the termination of S ($\mathit{trm}(S)$) and guarantee P (whatever non-determinism is involved in S), its complement $\neg\, [S]\neg\, P$ yields the weakest property needed to guarantee the feasibility of S ($\mathit{fis}(S)$ – its firing condition) and possibly establish P.

To conclude this abstract, we have shown how B's Generalised Substitution Language can be used to reason about parallel communicating (and shared variable) programs. Using process albegra's (here CCS') symbolic operational semantic approach we have acknowledged that parallel programs can be equated to sequential programs with goto control structures (characterised as labelled transition systems). GSL commands associated with the arcs of each transition allow us to reason in the weakest precondition style about what such programs achieve. To illustrate we gave two simple healthiness conditions. The full paper will contain additional examples of healthiness conditions and be illustrated with (a) working example(s).

References

[Abr96] J-R Abrial. *The B Book – Assigning Programs to Meanings.* Cambridge University Press, 1996.

[HJ98] C. A. R. Hoare and He J. *Unifying Theories of Programming.* Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.

[HL95] M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, (138), 1995.

[Lin96] H. Lin. Symbolic Bisimulation with Assignment. In *Proceedings of CONCUR 96*, number 1119 in Lecture Notes in Computer Science. Springer-Verlag, 1996.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, Hemel Hempstead, 1989.

[RH96] J. Rathke and M. Hennessy. Local model checking for a value-based modal μ-calculus. Technical Report 96:05, Department of Cognitive and Computing Sciences, University of Sussex., 1996.

[Spi95] M. Spivey. *The Z Reference Manual – 2nd edition.* Prentice-Hall, 1995.

Observational Semantics for Timed Event Structures*

Irina B. Virbitskaite

A. P. Ershov Institute of Informatics Systems
Siberian Division of the Russian Academy of Sciences
6, Acad. Lavrentiev ave., 630090, Novosibirsk, Russia
e-mail: virb@iis.nsk.su

Abstract. The paper is contributed to develop a family of observational equivalences for timed true concurrent models. In particular, we introduce three different semantics (sequences of actions, sequences of multisets, partial ordering of actions) for trace and bisimulation equivalences in the setting of event structures with dense time domain. We study the relationship between these three approaches and show their discriminating power. Furthermore, when dealing with particular subclasses of the model under consideration such as timed sequential and timed deterministic event structures there is no difference between a more concrete or a more abstract approach.

1 Introduction

An important ingredient of every theory of concurrency is a notion of equivalence between processes. Over the past several years, a variety of equivalences have been promoted, and the relationship between them has been quite well-understood (see, for example, [5]). Two main lines which have been followed there can be sketched as follows. The first aspect which is most dominant in the classical concurrency approaches is the so-called linear time — branching time spectrum. Here different possibilities are discussed to what extent the points of choice between different executions of systems are taken into account. In the linear time approach, a system is equated with the set of its possible executions (*trace equivalence*), i.e. points of choice are ignored. At the other end of the spectrum, *bisimulation equivalence* considers choices very precisely. The other aspect to follow is whether causalities between action occurrences are taken into account. In the interleaving approach, these are neglected. Using more expressive system models like Petri nets or event structures, causality based equivalences can be easily defined.

Those equivalences were considered for formal system models without time delays. Recently, a growing interest can be observed in modelling real-time systems which imply a need of a representation of the lapse of time. Several formal methods for specifying and reasoning about such systems have been proposed in the last ten years (see [1] as a survey). Whereas, the incorporation of real time into equivalence notions is less advanced. There are a few papers (see, for example, [4, 10, 13]) where decidability questions of time-sensitive equivalences are investigated. In the above-mentioned studies, real-time systems are represented by timed interleaving models — parallel timer processes or timed automata, containing fictitious time measuring elements called clocks.

In this paper, we seek to develop a framework for observational equivalences in the setting of a timed true concurrent model. In particular, we introduce three different semantics (sequences of actions, sequences of multisets, partial ordering of actions) for trace and bisimulation equivalences in the setting of event structures with dense time domain. This allows us to take into account processes' timing behaviour in addition to their degrees of relative concurrency and nondeterminism. We also study the interrelations between these three approaches to the semantics of timed concurrent systems. Furthermore, when dealing with particular subclasses of the model such as timed sequential and timed nondeterministic processes there is no difference between a more concrete or a more abstract approach. This line of research is sometimes referred to as comparative concurrency semantics.

There have been several motivations for this work. One has been the papers [6, 7, 11] which have developed concurrent variants of different observational equivalences in the setting of event structures. A next origin of this study has been given by a number of papers (see [4, 10, 13] among others), which have extensively studied time-sensitive equivalence notions for interleaving models. However, to our best knowledge, the literature of timed true concurrent models has hitherto lacked such the equivalences. In this regard, the papers [2, 9] are a welcome exception, where different notions of timed testing have been treated in the framework of timed event

* This work is partially supported by the Russian Fund of Basic Research (Grant N 00-01-00898).

structures. Finally, another origin has been the paper [3] where equivalences based on step semantics have been investigated for a class of stochastic Petri nets with discrete time.

The rest of the paper is organized as follows. The basic notions concerning timed event structures are introduced in the next section. The definitions of three different semantics (sequences of actions, sequences of multisets, partial ordering of actions) of timed trace and bisimulaton equivalences are given in the following three sections, respectively. In section 6, we establish the interrelations between the equivalence notions in the setting of the model under consideration and its subclasses. Section 7 contains some conclusions and remarks on future works. Due to lack of the space, the proofs are omitted, they can be found in the full version of the paper.

2 Timed Event Structures

In this section, we introduce some basic notions and notations concerning timed event structures.

We first recall a notion of an event structure [12]. The main idea behind event structures is to view distributed computations as action occurrences, called events, together with a notion of causality dependency between events (which is reasonably characterized via a partial order). Moreover, in order to model nondeterminism, there is a notion of conflicting (mutually incompatible) events. A labelling function records which action an event corresponds to.

Let Act be a finite set of actions. A *(labelled) event structure* over Act is a 4-tuple $S = (E, \leq, \#, l)$, where E is a countable set of events; $\leq \subseteq E \times E$ is a partial order (the *causality relation*), satisfying the *principle of finite causes*: $\forall e \in E \diamond \{e' \in E \mid e' \leq e\}$ is finite; $\# \subseteq E \times E$ is a symmetric and irreflexive relation (the *conflict relation*), satisfying the *principle of conflict heredity*: $\forall e, e', e'' \in E \diamond e \# e' \leq e'' \Rightarrow e \# e''$; $l : E \longrightarrow Act$ is a labelling function.

For an event structure $S = (E, \leq, \#, l)$, we define $\smile = (E \times E) \setminus (\leq \cup \leq^{-1} \cup \#)$ (the *concurrency relation*); for $e, f \in E$, we let $e \#^1 f \Leftrightarrow e \# f \wedge (\forall e', f' \in E \diamond e' \leq e \wedge f' \leq f \wedge e' \# f' \Rightarrow e' = e \wedge f' = f)$ (the *immediate conflict*). For $C \subseteq E$, the *restriction* of S to C is defined as $S \lceil C = (C, \leq \cap (C \times C), \# \cap (C \times C), l\mid_C)$.

Let $C \subseteq E$. Then C is *left-closed* iff $\forall e, e' \in E \diamond e \in C \wedge e' \leq e \Rightarrow e' \in C$; C is *conflict-free* iff $\forall e, e' \in C \diamond \neg(e \# e')$; C is a *configuration of* S iff C is left-closed and conflict-free. Let $\mathcal{C}(S)$ denote the set of all finite configurations of S.

Next we present a model of timed event structures which are a timed extension of event structures by associating their events with timing constraints that indicate event occurrence times with regard to a global clock. An execution of a timed event structure is a *timed configuration*, consisting of the configuration and the timing function recording a global time moment at which events occur.

Before introducing the concept of a timed event structure, we need to define some auxiliary notations. Let $\mathbf{N}$ be the set of natural numbers, and $\mathbf{R}_0^+$ the set of nonnegative real numbers. Define the set of intervals: $Interv = \{[d_1, d_2] \subset \mathbf{R}_0^+\}$.

We are now ready to introduce the concept of timed event structures.

Definition 1 *A* (labelled) timed event structure *over Act is a pair* $TS = (S, D)$, *where* $S = (E, \leq, \#, l)$ *is a (labelled) event structure over Act and* $D : E \longrightarrow Interv$ *is a timing function such that* $e' \leq_{TS} e \Rightarrow \min D(e') \leq \min D(e)$ *and* $\max D(e') \leq \max D(e)$.

In a graphic representation of a timed event structure, the corresponding action labels and time intervals are drawn near to events. If no confusion arises, we will often use action labels rather event identities to denote events. The <-relations are depicted by arcs (omitting those derivable by transitivity), and conflicts are also drawn (omitting those derivable by conflict heredity). Following these conventions, a trivial example of a labelled timed event structure is shown in Fig. 1.

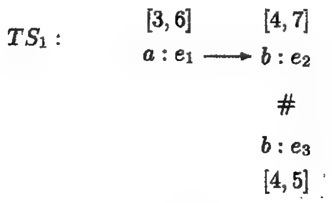


Fig. 1.

Timed event structures TS and TS' are *isomorphic* (denoted $TS \simeq TS'$), if there exists a bijection $\varphi : E_{TS} \longrightarrow E_{TS'}$ such that $e \leq_{TS} e' \iff \varphi(e) \leq_{TS'} \varphi(e')$, $e \#_{TS} e' \iff \varphi(e) \#_{TS'} \varphi(e')$, $l_{TS}(e) = l_{TS'}(\varphi(e))$, and $D_{TS}(e) = D_{TS'}(\varphi(e))$, for all $e, e' \in E_{TS}$.

Definition 2 *Let $TS = (S, D)$ be a timed event structure, $C \in \mathcal{C}(S)$, and $T : C \longrightarrow \{D(e) \mid e \in C\}$. Then $TC = (C, T)$ is a* timed configuration *of TS iff the following conditions hold:*

(i) $\forall\, e \in C \,\circ\, T(e) \in D(e)$;
(ii) $\forall\, e, e' \in C \,\circ\, e \leq_{TS} e' \Rightarrow T(e) \leq T(e')$;
(iii) $\forall\, e \in (E \setminus C) \,\circ$ (max $D(e) \geq T(e')$ *for all* $e' \in C$) *or*
(max $D(e) \geq T(e')$ *for some* $e' \in C$ *s.t.* $e' \# e$).

Informally speaking, a timed configuration consisting of the configuration and the timing function recording a global time moment at which events occur, satisfies the following requirements:

(i) an event can occur at a time when its timing constraints are met;
(ii) for all events e and e' occurred if e causally precedes e' then e should temporally precede e';
(iii) occurrences of events should not prevent other events to occur except for the events whose conflicting events have occurred before the events had time to occur.

The *initial timed configuration* of TS is $TC_{TS} = (\emptyset, 0)$. We use $\mathcal{TC}(TS)$ to denote the set of timed configurations of TS.

To illustrate the concept, consider all the possible timed configurations of the timed event structure TS_1 shown in Fig. 1: $(\emptyset, 0)$, $(\{e_1\}, T_1)$, $(\{e_3\}, T_2)$, $(\{e_1, e_3\}, T_3)$, $(\{e_1, e_2\}, T_4)$, where $T_1(e_1) \in [3, 5]$; $T_2(e_3) \in [4, 5]$; $T_3(e_1) \in [3, 6]$, $T_3(e_3) \in [4, 5]$; $T_4(e_1) \in [3, 5]$, $T_4(e_2) \in [4, 5]$, $T_4(e_1) \leq T_4(e_2)$.

From now on, for $TC_1 = (C_1, T_1), TC_2 = (C_2, T_2) \in \mathcal{TC}(TS)$ we shall write $TC_1 \longrightarrow TC_2$ iff $C_1 \subseteq C_2$, $T_2|_{C_1} = T_1$, and $\forall e \in C\ \forall e' \in (C_2 \setminus C_1) \circ T_1(e) \leq T_2(e')$.

3 Interleaving Semantics

In this section, we define timed trace and timed bisimulation equivalences based on an interleaving observation on timed event structures.

For this purpose we need the following notation. Let $(Act, \mathbf{R}_0^+) = \{(a, d) \mid a \in Act,\ d \in \mathbf{R}_0^+\}$ be the set of *timed actions.*

In the interleaving semantics, a timed event structure progresses through a sequence of timed configurations by occurrences of timed actions. In a timed configuration $TC_1 = (C_1, T_1)$, an *occurrence* of a timed action (a, d) *leads to* a timed configuration $TC_2 = (C_2, T_2)$ (denoted $TC_1 \xrightarrow{(a,d)} TC_2$), if $TC_1 \longrightarrow TC_2$, $C_2 = C_1 \cup \{e\}$, $l(e) = a$, and $T_2(e) = d$. The leading relation is extended to a sequence of timed actions from $(Act, \mathbf{R}_0^+)^*$ as follows: $TC \xrightarrow{(a_1,d_1)} \dots \xrightarrow{(a_n,d_n)} TC' \Leftrightarrow TC \xrightarrow{(a_1,d_1)\dots(a_n,d_n)} TC'$. The set $L_{ti}(TS) = \{w \in (Act, \mathbf{R}_0^+)^* \mid TC_{TS} \xrightarrow{w} TC$ for some $TC \in \mathcal{TC}(TS)\}$ is the *ti-language* of TS. As an illustration, consider the *ti*-language of the timed event structure TS_1, shown in Fig. 1: $\{\epsilon,\ (a, d_1),\ (b, d_2),\ (a, d_3)(b, d_4),\ (b, d_5)(a, d_6) \mid d_1, d_3 \in [3, 5],\ d_2, d_4, d_5 \in [4, 5],\ d_6 \in [4, 6],\ d_3 \leq d_4,\ d_5 \leq d_6\}$.

Definition 3 *Let TS and TS' be timed event structures.*

- *TS and TS' are* timed interleaving trace equivalent *(denoted $TS \equiv_{ti} TS'$) iff $L_{ti}(TS) = L_{ti}(TS')$, i.e., two timed event structures are timed interleaving trace equivalent, iff their ti-languages coincide;*
- *TS and TS' are* timed interleaving bisimilar *(denoted $TS \underline{\leftrightarrow}_{ti} TS'$) iff there exists a relation $\mathcal{B} \subseteq \mathcal{TC}(TS) \times \mathcal{TC}(TS')$ satisfying the following conditions: $(TC_{TS}, TC_{TS'}) \in \mathcal{B}$ and for all $(TC, TC') \in \mathcal{B}$ it holds:*
  (a) if $TC \xrightarrow{(a,d)} TC_1$ in TS, then $TC' \xrightarrow{(a,d)} TC_1'$ in TS' and $(TC_1, TC_1') \in \mathcal{B}$ for some $TC_1' \in \mathcal{TC}(TS')$,
  (b) if $TC' \xrightarrow{(a,d)} TC_1'$ in TS', then $TC \xrightarrow{(a,d)} TC_1$ in TS and $(TC_1, TC_1') \in \mathcal{B}$ for some $TC_1 \in \mathcal{TC}(TS)$,
  i.e., two timed event structures are timed interleaving bisimilar, if there exists a relation between their bisimilar timed configurations, among which the initial ones, such that the timed configurations obtained by occurring timed actions are also timed interleaving bisimilar.

Considering the timed event structures shown in Fig. 2, we get $TS_3 \equiv_{ti} TS_4$, while $TS_2 \not\equiv_{ti} TS_3$, since, for example, $(b, 0)(a, 0) \in L_{ti}(TS_3)$ and $(b, 0)(a, 0) \notin L_{ti}(TS_2)$. Further, we have $TS_4 \underline{\leftrightarrow}_{ti} TS_5$ but $TS_3 \not\underline{\leftrightarrow}_{ti} TS_4$, because in the timed configuration of TS_4, containing only the timed action $(a, 1)$, an occurrence of the timed action $(b, 1)$ is possible, but it is not always the case in TS_3.

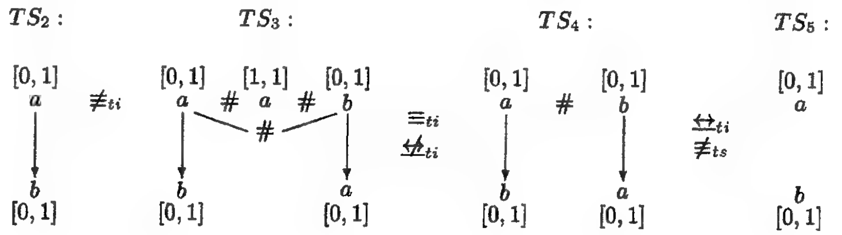


Fig. 2.

4 Step Semantics

In this section, we define a step observation on timed event structures to develop timed step trace and timed step bisimulation equivalences. Step semantics generalizes interleaving semantics by allowing timed actions to occur concurrently with themselves. We show that timed step semantics gives a more precise account of concurrency than the timed interleaving one.

Let A be an arbitrary set. A finite multiset M over A is a function $M : A \longrightarrow \mathbf{N}$ such that $| \{a \in A \mid M(a) > 0\} | < \infty$. Let $\mathcal{M}^{Act}$ to denote the set of finite nonempty multisets over Act. We use $(\mathcal{M}^{Act}, \mathbf{R}_0^+) = \{(A, d) \mid A \in \mathcal{M}^{Act},\ d \in \mathbf{R}_0^+\}$ to indicate the set of *timed steps.*

In the step semantics, timed configurations of a timed event structure change, if timed steps from $(\mathcal{M}^{Act}, \mathbf{R}_0^+)$ are executed. In a timed configuration $TC_1 = (C_1, T_1)$, an *execution* of a timed step $(A, d) \in (\mathcal{M}^{Act}, \mathbf{R}_0^+)$ *leads* to a timed configuration $TC_2 = (C_2, T_2)$ (denoted $TC_1 \xrightarrow{(A,d)} TC_2$), if $TC_1 \longrightarrow TC_2,\ C_2 \setminus C_1 = X,\ \forall\, e, e' \in X \circ e \smile e',\ l(X) = A,\ \forall e \in X \circ T_2(e) = d$, where $l(X)(a) = |\{e \in X \mid l(e) = a\}|$. The leading relation is extended to a sequence of timed steps from $(\mathcal{M}^{Act}, \mathbf{R}_0^+)^*$ as follows: $TC \xrightarrow{(A_1,d_1)} \dots \xrightarrow{(A_n,d_n)} TC' \Leftrightarrow TC \xrightarrow{(A_1,d_1)\dots(A_n,d_n)} TC'$. The set $L_{ts}(TS) = \{w \in (\mathcal{M}^{Act}, \mathbf{R}_0^+)^* \mid TC_{TS} \xrightarrow{w} TC \text{ for some } TC \in \mathcal{TC}(TS)\}$ is the *ts-language* of TS. Considering the timed event structure TS_1, shown in Fig. 1, we have $L_{ts}(TS) = \{\epsilon,\ (\{a\}, d_1),\ (\{b\}, d_2),\ (\{a\}, d_3)(\{b\}, d_4),\ (\{b\}, d_5)(\{a\}, d_6),\ (\{a, b\}, d_2) \mid d_1, d_3 \in [3, 5],\ d_2, d_4, d_5 \in [4, 5],\ d_6 \in [4, 6],\ d_3 \leq d_4,\ d_5 \leq d_6\}$.

Using ts-languages and leading relations of the form $\xrightarrow{(A,d)}$, we obtain timed step trace equivalence, $\equiv_{ts}$, and timed step bisimulation equivalence, $\underline{\leftrightarrow}_{ts}$, exactly as the corresponding interleaving equivalences in Definition 3. Timed step bisimulation is clearly stronger than both timed interleaving bisimulation and timed step trace equivalence.

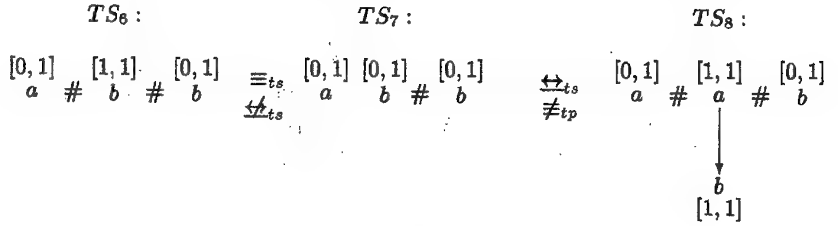


Fig. 3.

To illustrate the concepts, consider the timed event structures, shown in Fig. 2 and 3. We have $TS_6 \equiv_{ts} TS_7$, while $TS_4 \not\equiv_{ts} TS_5$, since, for example, $(\{a, b\}, 0) \in L_{ts}(TS_5)$ and $(\{a, b\}, 0) \notin L_{ts}(TS_4)$. Further, we get $TS_7 \underline{\leftrightarrow}_{ts} TS_8$ but $TS_6 \not\underline{\leftrightarrow}_{ts} TS_7$, because in a timed configuration of TS_7, containing only the timed action $(b, 1)$, an execution of the timed step $(\{a\}, 1)$ is always possible, but it is not the case in TS_6.

5 Partial Order Semantics

In this section, we consider several suggestions to define timed equivalence notions based on partial orders which take into account causality between timed actions.

Define a *timed partial order set* as a timed event structure $TP = (S_{TP} = (E_{TP}, \leq_{TP}, \#_{TP}, l_{TP}), D_{TP})$ such that $\#_{TP} = \emptyset$ and $D_{TP} : E_{TP} \longrightarrow Points$, where $Points = \{[d_1, d_2] \in Interv \mid d_1 = d_2\}$. Isomorphism classes of timed partial order sets are called *timed pomsets.*

We now consider leading relations of the form $\xrightarrow{TP}$, where TP is a timed pomset. For $TC_1 = (C_1, T_1), TC_2 = (C_2, T_2) \in \mathcal{TC}(TS)$, we shall write $TC_1 \xrightarrow{TP} TC_2$, if $TC_1 \longrightarrow TC_2$ and TP is the isomorphism class of $(S_{TS} \lceil (C_2 \setminus C_1)\ , T_2|_{(C_2 \setminus C_1)})$. The set $L_{tp}(TS) = \{TP \mid TC_{TS} \xrightarrow{TP} TC$ for some $TC \in \mathcal{TC}(TS)\}$ is the *tp-language* of TS. To illustrate the concept, we consider the *tp*-language of the timed event structure TS_1, shown in Fig. 1:

$$L_{tp}(TS_1) = \{\epsilon, \overset{[d_1,d_1]}{a}, \overset{[d_2,d_2]}{b}, \begin{matrix} \overset{[d_3,d_3]}{a} \\ \overset{[d_2,d_2]}{b} \end{matrix}, \overset{[d_4,d_4]}{a} \longrightarrow \overset{[d_5,d_5]}{b} \mid d_1, d_4 \in [3,5],\ d_2, d_5 \in [4,5],\ d_3 \in [3,6],\ d_4 \leq d_5\}.$$

Using *tp*-languages and leading relations of the form $\xrightarrow{TP}$, we obtain timed pomset trace equivalence, $\equiv_{tp}$, and timed pomset bisimulation equivalence, $\underline{\leftrightarrow}_{tp}$, exactly as the corresponding equivalences in Definition 3. Timed pomset bisimulation is clearly stronger than both timed step bisimulation and timed pomset trace equivalence.

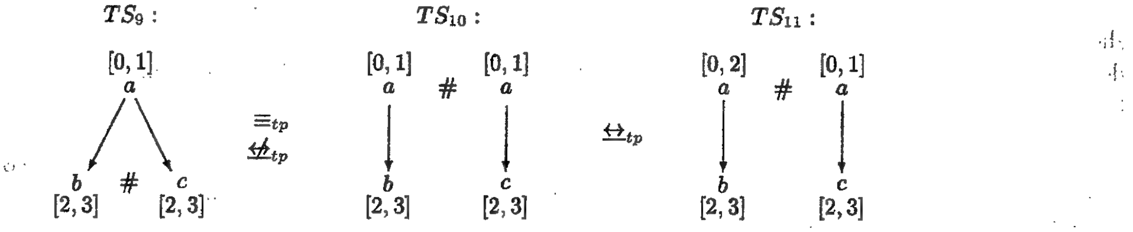


Fig. 4.

Considering the timed event structures, shown in Fig. 3 and 4, we obtain $TS_9 \equiv_{tp} TS_{10}$, while $TS_7 \not\equiv_{tp} TS_8$, since, for example, $\overset{[1,1]}{a} \longrightarrow \overset{[1,1]}{b} \in L_{tp}(TS_8)$ and $\overset{[1,1]}{a} \longrightarrow \overset{[1,1]}{b} \notin L_{tp}(TS_7)$. Further, we have $TS_{10} \underline{\leftrightarrow}_{tp} TS_{11}$, but $TS_9 \underline{\not\leftrightarrow}_{tp} TS_{10}$, because in the timed configuration of TS_9, containing only the timed action $(a, 1)$, the executions of both the timed pomset $\overset{[2,2]}{b}$ and the timed pomset $\overset{[2,2]}{c}$ are possible, but it is not the case in TS_{10}.

6 Comparison of Equivalences

The common framework used to define different observational equivalences allows us to study the relationships between the three induced semantics. The theorems we state in the section are a step towards a better understanding of the interrelations between interleaving, multisets, and partial order semantics. In particular, we will give the hierarchy for the equivalences and will establish that some of them coincide on particular subclasses of timed event structures.

Theorem 1 *Let TS and TS' be timed event structures. Then*

(i) $TS \equiv_{ti} TS' \Leftarrow TS \equiv_{ts} TS' \Leftarrow TS \equiv_{tp} TS'$;
(ii) $TS \underline{\leftrightarrow}_{ti} TS' \Leftarrow TS \underline{\leftrightarrow}_{ts} TS' \Leftarrow TS \underline{\leftrightarrow}_{tp} TS'$.

The timed event structures shown in Fig. 1-3 show that the converse implications of the above theorem do not hold and that the six equivalences are all different.

Now one can ask the obvious question what happens with all these equivalences if we restrict ourselves to some subclasses of the model under consideration. A timed event structure TS is called *sequential*, if $\smile_{TS} = \emptyset$; TS is *deterministic*, if $e \smile_{TS} e'$ or $e \#^1_{TS} e' \Rightarrow l(e) \neq_{TS} l(e')$ and $D_{TS}(e) \cap D_{TS}(e') \neq \emptyset$.

The next theorem shows that if we only consider timed event structures which represent timed sequential processes then all the three semantics of timed trace and timed bisimulation equivalences coincide.

Theorem 2 *Let TS and TS' be timed sequential event structures. Then*

(i) $TS \equiv_{ti} TS' \Rightarrow TS \equiv_{ts} TS' \Rightarrow TS \equiv_{tp} TS'$;
(ii) $TS \underline{\leftrightarrow}_{ti} TS' \Rightarrow TS \underline{\leftrightarrow}_{ts} TS' \Rightarrow TS \underline{\leftrightarrow}_{tp} TS'$.

The theorem below establishes that if we only consider timed event structures which represent timed deterministic processes then timed step and timed partial order semantics coincide.

Theorem 3 *Let TS and TS' be timed deterministic event structures. Then*

(i) $TS \equiv_{ts} TS' \Rightarrow TS \equiv_{tp} TS'$;
(ii) $TS \underline{\leftrightarrow}_{ts} TS' \Rightarrow TS \underline{\leftrightarrow}_{tp} TS'$.

The two rightmost timed event structures in Fig. 2 show that even for timed deterministic event structures there is a difference between timed interleaving and timed partial order semantics.

7 Conclusion

In this paper, we have given a flexible abstract mechanism, based on observational equivalences which allows us to consider timed event structures as the basis of three different approaches (sequences of actions, sequences of multisets, partial ordering of actions) to the description of concurrent and real time systems. The results obtained show that these three semantics in general provide formal tools with an increasing power. Furthermore, when dealing with particular subclasses of the model such as timed sequential and timed deterministic processes there is no difference between a more concrete or a more abstract approach.

In a future work, we plan to extend the obtained results to other observational equivalences (e.g., equivalences taking into account internal actions) of timed systems. Some investigation on different timed concurrent semantics of testing equivalence which is located between trace and bisimulation equivalences in the linear time – branching time spectrum is now under way and we plan to report on this work elsewhere.

References

1. R. ALUR, T.A. HENZINGER. Logics and models of real time: a survey. *Lecture Notes in Computer Science* **600** (1992) 74–106.
2. M.V. ANDREEVA, E.N. BOZHENKOVA, I.B. VIRBITSKAITE. Analysis of timed concurrent models based on testing equivalence. *Fundamenta Informaticae* **43(1-4)**(2000) 1–20.
3. P. BUCHHOLZ, I. TARASYUK. A class of stochastic Petri nets with step semantics and related equivalence notions. Technische Berichte TUD-FI00-12, Technische Universitaet Dresden (2000).
4. K. ČERĀNS. Decidability of bisimulation equivalences for parallel timer processes. *Lecture Notes in Computer Science* **663** (1993) 302–315.
5. R.J. VAN GLABBEEK. The linear time – branching time spectrum II: the semantics of sequential systems with silent moves. Extended abstract. *Lecture Notes in Computer Science* **715** (1993) 66–81.
6. R.J. VAN GLABBEEK, U. GOLTZ. Equivalence notions for concurrent systems and refinement of actions. *Lecture Notes in Computer Science* **379** (1989) 237–248.
7. U. GOLTZ, H. WEHRHEIM. Causal testing. *Lecture Notes in Computer Science* **1113** (1996) 394–406.
8. J.-P. KATOEN, R. LANGERAK, D. LATELLA, E. BRINKSMA. On specifying real-time systems in a causality-based setting. *Lecture Notes in Computer Science* **1135** (1996) 385–404.
9. D. MURPHY. Time and duration in noninterleaving concurrency. *Fundamenta Informaticae* **19** (1993) 403–416.
10. B. STEFFEN, C. WEISE. Deciding testing equivalence for real-time processes with dense time. *Lecture Notes in Computer Science* **711** (1993) 703–713.
11. F.W. VAANDRAGER. Determinism → (event structure isomorphism = step sequence equivalence). *Theoretical Computer Science* **79(2)** (1991) 275–294.
12. G. WINSKEL. An introduction to event structures. *Lecture Notes in Computer Science* **354** (1988) 364–397.
13. C. WEISE, D. LENZKES. Efficient scaling-invariant checking of timed bisimulation. *Lecture Notes in Computer Science* **1200** (1997) 176–188.

The Impact of Synchronisation on Secure Information Flow in Concurrent Programs

Andrei Sabelfeld

Department of Computer Science, Chalmers University of Technology
and University of Göteborg, 412 96 Göteborg, Sweden
fax: +46 31 16 56 55, e-mail: andrei@cs.chalmers.se

Abstract. Synchronisation is fundamental to concurrent programs. This paper investigates confidentiality for multi-threaded programs in the presence of synchronisation. We give a small-step operational semantics for a simple shared-memory multi-threaded language with synchronisation and present a compositional timing-sensitive bisimulation-based confidentiality specification. We propose a type-based analysis improving on previous approaches to reject potentially insecure programs.

1 Introduction

Motivation. This paper focuses on the problem of program *confidentiality*, i.e., determining whether a given shared-memory multi-threaded program has secure information flow. The program runs on a partition of data on high (private) and low (public) security data, although a more general lattice of security levels can be considered. The program is not trusted (possibly received over the Internet). The program's low output is publicly available (e.g., sent over the Internet) as well as, possibly, timing information about the program's execution (e.g., times when the program makes Internet accesses are observable).

Background. The problem of confidentiality for various programming languages has been investigated by many researchers including [7, 8, 6, 3, 13, 5, 14, 21, 10, 19, 1, 20, 11, 16, 2, 17]. The issue of secure information flow has become especially important with the growing popularity of mobile code and networked information systems. For distributed programming, the use of multi-threaded programming languages has become extremely popular [4]. However, in the setting of an imperative shared-memory multi-threaded language, the majority of investigations in the area of secure information flow, e.g., [19, 20, 16] do not have *synchronisation* in the language. Although the security logic of [3] does treat synchronisation primitives, there is neither a soundness proof nor a decision algorithm given for the logic. Because synchronisation is fundamental to concurrent programs, it is highly desirable to have a robust security specification and tools that aid in the design of secure programs with synchronisation. To bridge the gap, this paper presents a compositional bisimulation-based confidentiality specification for multi-threaded programs with synchronisation and proposes a type-based analysis improving on previous approaches to reject potentially insecure programs.

Insecure Flows to Eliminate. Let us exemplify the types of insecure information flow that are in the focus of this paper. Suppose h is a high security variable and l is a low security one. There are several ways to leak information from h to the attacker. An example of a *direct* flow is the simple program $l := h$. An instance of an *indirect* flow through branching on a high condition is if $h = 1$ then $l := 1$ else $l := 0$. From the timing behaviour of the program the attacker may deduce secret information. The program $l := 0$; if $h = 1$ then (while $l < \mathit{MaxInteger}$ do $l := l + 1$) else skip is an instance of a *timing* leak. The program if $h = 1$ then (while True do skip) else skip is a variation of the timing leak called a *termination* leak. Observing the termination of the program reveals that h was not 1. Blocking a thread can change the observable behaviour of a computation, e.g., its termination behaviour. If blocking depends on high data, then the attacker might learn secrets through the observable behaviour. Concrete examples of *synchronisation* leaks are postponed until Section 4 where concrete synchronisation primitives will be available.

Overview. The rest of the paper is organised as follows. Section 2 introduces the syntax and semantics of a multi-threaded language. Section 3 motivates and specifies a bisimulation-based security definition. Section 4 gives a type system for detecting secure programs and shows its correctness with respect to the security definition. Section 5 concludes by discussing related work.

2 A Multi-Threaded Language with Synchronisation

Semaphores are widely used as a synchronisation primitive in shared-memory languages. As pointed out by Andrews ([4], p.viii): *"Semaphores were the first high-level concurrent programming mechanism and remain one of the most important."* Semaphores are general enough to implement both *mutual exclusion* and *condition synchronisation*. In this section we introduce semaphores in the language and give a small-step operational semantics.

A *semaphore* [9] is a special variable (call it *sem*) that can only be manipulated by two commands: wait(*sem*) and signal(*sem*). The value of *sem* ranges over nonnegative integers. Initially, the value is 0 for every semaphore. The wait(*sem*) command blocks until *sem* is positive. Once *sem* is positive, it gets decremented by 1. The signal(*sem*) command increments *sem* by 1. One approach to introducing semaphores is defining the synchronisation primitives by while-loop-based *busy waiting* (as in, e.g., [3]):

$$\mathsf{wait}(sem) = (\mathsf{while}\ sem{=}0\ \mathsf{do\ skip}); sem := sem-1 \quad \mathsf{signal}(sem) = sem := sem+1$$

Although these definitions are intuitive, there are two major problems with them. First, a waiting process *occupies the CPU with idle spinning.* Also, *delay and decrement must be a single atomic action.* Otherwise two wait(*sem*) threads might succeed when the initial value of *sem* is 1! Atomicity may be hard to implement in a distributed setting (not to mention that timing behaviour will spin out of control since one atomic action no longer takes one time unit). *Blocked waiting*, which commonly underlies semaphore implementations, does not have the disadvantages above. It is important that we adapt blocked waiting, since the dynamics of thread (un)blocking in a program's execution needs to be explicitly modelled due to its potential to affect the program's security.

In order to define blocked waiting, we will use a special pool of waiting processes. The idea is that blocked processes should be sleeping (as opposed to spinning in busy waiting) until the respective signal is sent. The syntax of the language is given in Figure 1. Let $C, D, E, \ldots$ range over commands *Com*, and let $\boldsymbol{C}$ denote a vector of commands of the form $\langle C_1 \ldots C_n\rangle$. Vectors $\boldsymbol{C}, \boldsymbol{D}, \boldsymbol{E}, \ldots$ range over $\boldsymbol{Com} = \cup_{n\in\mathbb{N}} Com^n$, the set of thread pools (or programs). A *state* $s \in \mathbf{St}$ is a finite mapping from variables (including special semaphore variables) to values. The set of variables is partitioned into high and low security classes. h and l will denote typical high and low variables, respectively. Define *low-equivalence* by $s_1 =_L s_2$ iff the low components of s_1 and s_2 are the same. The small-step semantics is given by transitions between *configurations*, i.e., between pairs each containing a program and a state. The deterministic part of the semantics is defined by the transition rules in Figure 2. Arithmetic and boolean expressions are executed atomically by $\downarrow$ transitions. The general form of a deterministic $\rightharpoonup$ transition is either $\langle\!| C, s |\!\rangle \rightharpoonup \langle\!| \langle\rangle, s' |\!\rangle$, which means termination with the final state s', or $\langle\!| C, s |\!\rangle \rightharpoonup \langle\!| C'\boldsymbol{D}, s' |\!\rangle$. Here, one step of computation that starts with a command C in a state s gives a new main thread C', a (possibly empty) vector of spawned processes $\boldsymbol{D}$ and a new state s'.

Let *sem* be a variable from the special set *Sem* of semaphore variables. The wait(*sem*) command emits the "block" label $\otimes sem$ in case the value of *sem* is 0. Otherwise it decreases the value of *sem*. The signal(*sem*) emits the "unblock" label $\odot sem$. The labels are propagated through the sequential composition to the top level.

The concurrent part of the semantics is given in Figure 3. The nondeterministic $\rightarrow$ transitions are of the form $\langle\!| \boldsymbol{C}, w, s |\!\rangle \rightarrow \langle\!| \boldsymbol{C}', w', s' |\!\rangle$ where configurations are equipped with queues of waiting processes $w, w' : Sem \rightarrow \boldsymbol{Com}$. Whenever the top level receives a $\otimes sem$ signal, the blocked thread is put in the end of the FIFO queue associated with *sem*. If the top level receives an $\odot sem$ signal, the first thread in the FIFO gets awakened or, in case, the FIFO is empty, the value of *sem* gets incremented. Nondeterminism is resolved by the scheduler in a particular implementation.

3 Security Specification

What is a secure program in the language we have just defined? This section focuses motivating and defining confidentiality for our language.

The central idea of *extensional* security, as opposed to *intensional* security, is that confidentiality should not be specified by a special-purpose security formalism, but, rather, should be defined in terms of a standard semantics as a dependency property (more precisely, the absence of dependencies). If direct, indirect, and timing flows are considered, then, intuitively, a program has the extensional *noninterference* property, if *varying the high input will not change the possible low-level observations*, i.e., low inputs, low outputs and timing. Many investigations have successfully pursued the extensional view, including [6, 21, 10, 19, 1, 20, 11, 16, 2, 17] for the

$$\begin{aligned} C ::= {} & \mathsf{skip} \mid Id := Exp \mid C_1; C_2 \mid \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \\ & \mid \mathsf{while}\ B\ \mathsf{do}\ C \mid \mathsf{fork}(C, \boldsymbol{D}) \mid \mathsf{wait}(Sem) \mid \mathsf{signal}(Sem) \end{aligned}$$

Fig. 1. Command syntax

$$\langle\!| \mathsf{skip}, s |\!\rangle \rightarrow \langle\!| \langle\rangle, s |\!\rangle \qquad \frac{\langle\!| exp, s |\!\rangle \downarrow n}{\langle\!| x := exp, s |\!\rangle \rightarrow \langle\!| \langle\rangle, [x := n]s |\!\rangle}$$

$$\frac{\langle\!| C_1, s |\!\rangle \rightarrow \langle\!| \langle\rangle, s' |\!\rangle}{\langle\!| C_1; C_2, s |\!\rangle \rightarrow \langle\!| C_2, s' |\!\rangle} \qquad \frac{\langle\!| C_1, s |\!\rangle \rightarrow \langle\!| C_1' \boldsymbol{D}, s' |\!\rangle}{\langle\!| C_1; C_2, s |\!\rangle \rightarrow \langle\!| (C_1'; C_2) \boldsymbol{D}, s' |\!\rangle}$$

$$\frac{\langle\!| B, s |\!\rangle \downarrow \mathsf{True}}{\langle\!| \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s |\!\rangle \rightarrow \langle\!| C_1, s |\!\rangle} \qquad \frac{\langle\!| B, s |\!\rangle \downarrow \mathsf{False}}{\langle\!| \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s |\!\rangle \rightarrow \langle\!| C_2, s |\!\rangle}$$

$$\frac{\langle\!| B, s |\!\rangle \downarrow \mathsf{True}}{\langle\!| \mathsf{while}\ B\ \mathsf{do}\ C, s |\!\rangle \rightarrow \langle\!| C; \mathsf{while}\ B\ \mathsf{do}\ C, s |\!\rangle} \qquad \frac{\langle\!| B, s |\!\rangle \downarrow \mathsf{False}}{\langle\!| \mathsf{while}\ B\ \mathsf{do}\ C, s |\!\rangle \rightarrow \langle\!| \langle\rangle, s |\!\rangle}$$

$$\frac{\langle\!| sem, s |\!\rangle \downarrow n \quad n > 0}{\langle\!| \mathsf{wait}(sem), s |\!\rangle \rightarrow \langle\!| \langle\rangle, [sem := sem - 1]s |\!\rangle} \qquad \frac{\langle\!| sem, s |\!\rangle \downarrow 0}{\langle\!| \mathsf{wait}(sem), s |\!\rangle \overset{\otimes sem}{\rightarrow} \langle\!| \langle\rangle, s |\!\rangle}$$

$$\langle\!| \mathsf{fork}(C, \boldsymbol{D}), s |\!\rangle \rightarrow \langle\!| C\boldsymbol{D}, s |\!\rangle \qquad \frac{\langle\!| C_1, s |\!\rangle \overset{\alpha}{\rightarrow} \langle\!| C_1', s' |\!\rangle \quad \alpha \in \{\otimes sem, \odot sem\}}{\langle\!| C_1; C_2, s |\!\rangle \overset{\alpha}{\rightarrow} \langle\!| C_1'; C_2, s' |\!\rangle}$$

$$\langle\!| \mathsf{signal}(sem), s |\!\rangle \overset{\odot sem}{\rightarrow} \langle\!| \langle\rangle, s |\!\rangle$$

Fig. 2. Small-step deterministic semantics of commands

$$\frac{\langle\!| C_i, s |\!\rangle \rightarrow \langle\!| \boldsymbol{C}, s' |\!\rangle}{\langle\!| \langle C_1 \dots C_n \rangle, w, s |\!\rangle \rightarrow \langle\!| \langle C_1 \dots C_{i-1} \boldsymbol{C} C_{i+1} \dots C_n \rangle, w, s' |\!\rangle}$$

$$\frac{\langle\!| C_i, s |\!\rangle \overset{\otimes sem}{\rightarrow} \langle\!| C', s' |\!\rangle \qquad w_{sem} = \boldsymbol{D}}{\langle\!| \langle C_1 \dots C_n \rangle, w, s |\!\rangle \rightarrow \langle\!| \langle C_1 \dots C_{i-1} C_{i+1} \dots C_n \rangle, [w_{sem} := \boldsymbol{D} C']w, s' |\!\rangle}$$

$$\frac{\langle\!| C_i, s |\!\rangle \overset{\odot sem}{\rightarrow} \langle\!| C', s' |\!\rangle \qquad w_{sem} = C\boldsymbol{D}}{\langle\!| \langle C_1 \dots C_n \rangle, w, s |\!\rangle \rightarrow \langle\!| \langle C_1 \dots C_{i-1} C' C_{i+1} \dots C_n C \rangle, [w_{sem} := \boldsymbol{D}]w, s' |\!\rangle}$$

$$\frac{\langle\!| C_i, s |\!\rangle \overset{\odot sem}{\rightarrow} \langle\!| C', s' |\!\rangle \qquad w_{sem} = \langle\rangle}{\langle\!| \langle C_1 \dots C_n \rangle, w, s |\!\rangle \rightarrow \langle\!| \langle C_1 \dots C_{i-1} C' C_{i+1} \dots C_n \rangle, w, [sem := sem + 1]s' |\!\rangle}$$

Fig. 3. Concurrent semantics of thread pools

justification of security analyses and verification techniques for different languages. We follow the extensional approach and focus on the extensional security for our language.

The main idea behind the bisimulation-based approach promoted by [16] is to define a *low-bisimulation* on commands such that the indistinguishability of the behaviours of two programs C and D for the attacker is formalised by $C \sim_L D$. Here $\sim_L$ is an appropriate low-bisimulation that may be flexibly tuned depending on a specific degree of security. For a given low-bisimulation $\sim_L$, the definition of security is simply: "C is secure iff $C \sim_L C$". For the purpose of this paper we adapt a variation of the *strong low-bisimulation* [16]. Let $\alpha \in \{\epsilon, \otimes sem, \odot sem\}$.

Definition 1 *Define the* strong *low-bisimulation* $\cong_L$ *to be the union of all symmetric relations R on thread pools of equal size, such that if $\langle C_1 \dots C_n \rangle\ R\ \langle D_1 \dots D_n \rangle$ then for all i, α, s_1 and s_2 (such that $s_1 =_L s_2$)*

$$\langle\!| C_i, s_1 |\!\rangle \overset{\alpha}{\rightarrow} \langle\!| \boldsymbol{C}', s_1' |\!\rangle \Longrightarrow \exists \boldsymbol{D}', s_2'. \langle\!| D_i, s_2 |\!\rangle \overset{\alpha}{\rightarrow} \langle\!| \boldsymbol{D}', s_2' |\!\rangle, \boldsymbol{C}'\ R\ \boldsymbol{D}', s_1' =_L s_2'$$

Definition 2 $\boldsymbol{C}$ *is* secure $\Longleftrightarrow \boldsymbol{C} \cong_L \boldsymbol{C}$

One can show that the relation $\cong_L$ is transitive, but not reflexive. E.g., $l := h \ncong_L l := h$ which reflects the fact that the program behaves differently for the attacker, depending on the initial value of h. The choice of this bisimulation allows for robust security. As argued in [16], ***strong low-bisimulation captures timing flows.*** If two commands may have a different timing behaviour depending on high data (which would result in information flow from high to low) then they are not low-bisimilar. Also, ***strong low-bisimulation is scheduler-independent.*** Thus, our notion of security is robust with respect to any choice of a particular scheduler.

4 Type-Based Security Analysis

This section presents an automatic compositional analysis for certifying secure programs, extending and improving on previous approaches [3, 20, 2, 16].

The Type System. The analysis is based on a type system that transforms a given program into a new program. If the initial program is free of direct, indirect and synchronisation insecure information flows then it might be accepted by the system and transformed into a program that is also free of timing leaks. Otherwise the initial program is rejected. The transformation rules have the form $\boldsymbol{C \hookrightarrow C' : Sl}$, where $\boldsymbol{C}$ is a program, $\boldsymbol{C'}$ is the result of its transformation and $\boldsymbol{Sl}$ is the type of $\boldsymbol{C'}$. The type $\boldsymbol{Sl}$ is $\boldsymbol{C'}$'s *low slice*, i.e., essentially a copy of $\boldsymbol{C'}$ in which assignments to (and conditionals on) high variables have been replaced by skip's. The low slice $\boldsymbol{Sl}$ has no occurrences of h and models the timing behaviour of C', as observable by other threads.

We sketch the important typing and transformation rules. The complete set of the rules can be found in Chapter 4 of [15]. The variables h and l have the types *high* and *low* respectively. Value literals n may be considered as either *high* or *low*. An arbitrary expression *Exp* may be considered as *high*. An expression is typed *low* if it is composed from expressions typed *low*. Command skip is its own low slice and therefore its own type: skip $\hookrightarrow$ skip : skip. An assignment to a high variable is typed with the low slice skip, i.e., $h := Exp \hookrightarrow h := Exp : \mathsf{skip}$. The rule for an assignment to a low variable prevents direct insecure information flows (e.g., the assignment $l := h$ is not typable):

$$\frac{Exp : low}{l := Exp \hookrightarrow l := Exp : l := Exp} \qquad \frac{B : low \quad C \hookrightarrow C' : Sl}{\mathsf{while}\ B\ \mathsf{do}\ C \hookrightarrow \mathsf{while}\ B\ \mathsf{do}\ C' : \mathsf{while}\ B\ \mathsf{do}\ Sl}$$

The guard of the while-loop has to be low in order to prevent the timing (and nontermination) flow from the loop's guard. In the rules for fork, if (on a low condition), sequential and parallel composition, the transformed program is constructed compositionally using the same constructs as the original program. In addition to the insecure flows exemplified in Section 1, there are further ways to leak information through blocking. Synchronising on a high semaphore variable leads to (un)blocking of a thread and, clearly, may affect the termination behaviour of a program. As a consequence, neither wait(*sem*) nor signal(*sem*) on a high semaphore is allowed by the type system:

$$\frac{sem : low}{\mathsf{wait}(sem) \hookrightarrow \mathsf{wait}(sem) : \mathsf{wait}(sem)} \qquad \frac{sem : low}{\mathsf{signal}(sem) \hookrightarrow \mathsf{signal}(sem) : \mathsf{signal}(sem)}$$

The rule for an if (on a high conditional) prevents indirect insecure flows and timing flows. An indirect flow might be performed through synchronisation on a low semaphore depending on a high guard: if $h = 1$ then signal(*sem*) else skip. Thus, the type system disallows synchronisation in the branches of an if-on-high:

$$\frac{B : high \quad C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2 \quad al(Sl_1) = al(Sl_2) = \mathsf{False}}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}\ B\ \mathsf{then}\ C_1'; Sl_2\ \mathsf{else}\ Sl_1; C_2' : \mathsf{skip}; Sl_1; Sl_2}$$

where $al(C)$ is a boolean function returning True iff there is a syntactic occurrence of either an assignment to a low variable or a synchronisation primitive in C. The condition $al(Sl_1) = al(Sl_2) = \mathsf{False}$ prevents indirect leaks. Both branches must be typable (i.e., they must have a low slice). For the transformed program to be secure (preventing timing leaks) it is also necessary that the two branches be low-bisimilar. This is achieved by cross-copying the low slice of one branch into the other. The slice of the overall command is the sequential composition of the slices of the branches prefixed with a skip corresponding to the time tick for the guard inspection.

A modification of the rule above might be used to guarantee that dummy computation is inserted "evenly" and it does not block useful computation within the branches of the resulting program. Such a rule makes use of the parallel composition instead of the sequential one when cross-copying (assuming *sem* is a fresh semaphore,

Ref	NI	Aut	Sync	Time
Andrews&Reitman [3]	No	No	Yes	No
Volpano&Smith [20]	Yes	Yes	No	protect
Agat [2]	Yes	Yes	No	Yes
Sabelfeld&Sands [16]	Yes	Yes	No	Yes
This paper	Yes	Yes	Yes	Yes

Fig. 4. Approaches to security analysis of multi-threaded programs

unused anywhere in the program):

$$\frac{B : \mathit{high} \quad C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2 \quad al(Sl_1) = al(Sl_2) = \mathsf{False}}{\begin{array}{c}\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}\ B\ \mathsf{then}\ \mathsf{fork}(C_1'; \mathsf{wait}(sem), Sl_2; \mathsf{signal}(sem))\ \mathsf{else}\\ \mathsf{fork}(Sl_1; \mathsf{wait}(sem), C_2'; \mathsf{signal}(sem)) : \mathsf{skip}; \mathsf{fork}(Sl_1; \mathsf{wait}(sem), Sl_2; \mathsf{signal}(sem))\end{array}}$$

Despite the restrictions imposed by security, one can still write useful programs. We refer to [15] for an example of secure programming with synchronisation (implementing simple web-form processing) accepted by the type system.

Correctness of the Analysis. The key to straightforward correctness proofs is the *compositionality* of the security property (Definition 2). In the standard security terminology, this is called the *hook-up* property [12], which facilitates modular development of secure code. Since both the security property and the analysis are compositional, the correctness proof is a simple structural induction. For the compositionality proofs of the security property we refer to [15].

Theorem 1 $C \hookrightarrow C' : Sl \Longrightarrow C' \approxeq_L Sl$.

Corollary 1 (Correctness of the Analysis) $C \hookrightarrow C' : Sl \Longrightarrow C'$ *is secure.*

Soundness of the Transformation. We have shown that the result of the transformation is secure, but what of its relation to the original program? Clearly, the padding introduced by the transformation can change the timing, but otherwise it is just additional "stuttering". To make this point precise, let us define a *weak (bi)simulation* on configurations. Let $\langle C, s\rangle \to^{\star} \langle C', s'\rangle$ hold iff $\langle C, s\rangle = \langle C', s'\rangle$ or $\langle C, s\rangle \to \langle C', s'\rangle$.

Definition 3 *Define the weak simulation $\preceq$ (resp., bisimulation $\simeq$) to be the union of all (resp., symmetric) relations R on thread pools such that if $\vec{C}\ R\ \vec{D}$ and $\forall sem. w_{sem}\ R\ v_{sem}$ then for all s, s', w', $\vec{C}'$, there exists $\vec{D}'$ and v' such that*

$$\langle \vec{C}, w, s\rangle \to \langle \vec{C}', w', s'\rangle \Longrightarrow \langle \vec{D}, v, s\rangle \to^{\star} \langle \vec{D}', v', s'\rangle, \vec{C}'\ R\ \vec{D}', \forall sem. w'_{sem}\ R\ v'_{sem}$$

Theorem 2 $C \hookrightarrow C' : Sl \Longrightarrow C' \preceq C$.

The proof is by induction on the height of the transformation derivation. We refer to [15] for details.

5 Conclusions and Related Work

We have investigated the security of information flow in multi-threaded programs in the presence of synchronisation. The main result is that allowing neither synchronisation on high nor any synchronisation (not even on low) in the branches of an if-on-high is sufficient for building up a compositional timing-sensitive security specification from previous definitions that did not consider synchronisation. We have also proposed a type-based analysis that certifies programs according to the security specification. Let us conclude by discussing some improvements of this analysis compared to previous certification systems for security.

Figure 4 gives a comparative overview of some of the most related approaches to analysing confidentiality of multi-threaded programs. The first column **Ref** gives references to the related systems. **NI** means whether or not the system has been proved to certify programs to be secure under an extensional noninterference-like security property. The second and the fourth systems have been proved to guarantee *probabilistic noninterference.* Probabilistic noninterference seems to hold for the presented system. In fact, it has been designed with probabilistic noninterference in mind—future work includes a formal development of the soundness proof with respect to probabilistic noninterference. **Aut** stands for automatic. All but the first system are automatic analyses. The first system is formulated as a logic without any decision algorithms or soundness proofs. **Sync** indicates whether or not the underlying language has synchronisation primitives. Only the first and the present investigations consider synchronisation. Furthermore, Agat's study [2] only considers sequential programs.

Time says whether or not the system captures timing covert channels. Andrews and Reitman's paper [3] sketches the possibility of taking into account timing leaks in their security logic. However, the proposed mechanism rejects all programs that branch on high variables. Volpano and Smith do consider timing flows in [20]. They allow branching on high by requiring all if-on-high commands to be embraced by special protect commands. The protect executes atomically by definition, making the timing difference invisible for the attacker. Such a command seems difficult to implement without locking the execution of every atomic command in the language or, as suggested by Smith [18], using additional mechanisms like thread priorities. Even that will not close external timing leaks, i.e., a time-consuming computation will not be hidden by a protect from the attacker with a stop-watch.

Acknowledgements

Thanks are due to David Sands for fruitful discussions and to Johan Agat and Makoto Takeyama for many valuable comments.

References

1. M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *POPL '99, Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages (January 1999)*, 1999.
2. J. Agat. Transforming out timing leaks. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 40–53, January 2000.
3. G. R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM TOPLAS*, 2(1):56–75, January 1980.
4. Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison Wesley, 2000.
5. J.-P. Banatre, C. Bryce, and D. Le Metayer. Compile-time detection of information flow in sequential programs. *LNCS*, 875:55–73, 1994.
6. E. S. Cohen. Information transmission in sequential programs. In Richard A. DeMillo, David P. Dobkin, Anita K. Jones, and Richard J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.
7. D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.
8. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.
9. E.W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968.
10. N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Conference Record of the Twenty-Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 365–377, 1998.
11. K. R. M. Leino and Rajeev Joshi. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–138, 2000.
12. D. McCullough. Specifications for multi-level security and hook-up property. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 161–166. IEEE Computer Society Press, 1987.
13. M. Mizuno and D. Schmidt. A security flow control algorithm and its denotational semantics correctness proof. *Formal Aspects of Computing*, 4(6A):727–754, 1992.
14. P. Ørbæk. Can you Trust your Data? In *Proceedings of the TAPSOFT/FASE'95 Conference*, LNCS 915, pages 575–590, May 1995.
15. A. Sabelfeld. *Semantic Models for the Security of Sequential and Concurrent Programs*. PhD thesis, Chalmers University of Technology and Göteborg University, May 2001.
16. A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 200–214, Cambridge, England, July 2000.
17. A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, March 2001.
18. G. Smith. Personal communication, 2000.
19. G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 355–364, 19–21 January 1998.
20. D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2,3):231–253, November 1999.
21. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):1–21, 1996.

Dynamical Priorities without Time Measurement and Modification of the TCP

Valery A. Sokolov and Eugeny A. Timofeev

Yaroslavl State University,
150 000,Yaroslavl, Russia
e-mail: {sokolov, tim}@uniyar.ac.ru

Abstract. A priority discipline without time measurements is described. The universality of this discipline is proved. The application of this discipline for the performance of TCP is shown.

1 Introduction

In this paper we describe a dynamic (changing in time) priority discipline of a service system, which does not require time measurements. We prove that this service discipline has all possible queue mean lengths for a one-device service system with ordinary flows of primary customers and branching flows of secondary customers.

The use of the discipline for organizing the work of the TCP (Transmission Control Protocol) [1] allows us to eliminate some TCP disadvantages such as: the interpretation of a lost packet as the network overload, bulk transfer and so on. In this case, there is no need to make any changes in the TCP.

There are some works (see, for example, [3], [2]) which improve the TCP. The works [4], [5] propose a TCP modification — "TCP with an adaptive rate" (ARTCP), which improves the TCP, but requires the introduction of additional field to the protocol, namely the round trip time interval.

2 Theoretical Results

In this section we describe a service discipline (the rule of selecting packets) without time measerements. We call it a probabilistic-divided service discipline. For the one-device service system with ordinary flows of primary customers and branching flows of secondary customers we show that this discipline has all possible queue mean lengths.

2.1 Probabilistic-Divided Service Discipline

Now we introduce a probabilistic-divided service discipline [8].

Let our system have n types of customers. The parameters of this discipline are

- a cortege of $N = 2n$ natural numbers $\mathbf{a} = (a_1, a_2, \dots, a_N)$ such that

$$\forall i \in \{1, 2, \dots, n\} \quad \exists 1 \le f(i) < l(i) \le N : a_{f(i)} = a_{l(i)} = i; \tag{1}$$

- N real numbers $\mathbf{b} = (b_1, \dots, b_N)$ $(0 \le b_i)$, such that

$$b_{f(i)} + b_{l(i)} = 1 \quad \forall i \in \{1, 2, \dots, n\}. \tag{2}$$

Define the priority p_i $(1 \le p_i \le N)$ of every new customer of the type i $(1 \le i \le n)$ (arrival or appear after branching) as

$$p_i = \begin{cases} f(i), \text{ with the probability } & b_{f(i)}; \\ l(i), \text{ with the probability } & b_{l(i)}; \end{cases} \tag{3}$$

where $f(i), l(i)$ is defined in 1.

For the service we select the customer with the least priority.

2.2 Model

The service system has proven to be a useful tool for system performance analysis. In this section we have extended the application of such a system by incorporating populations of branching customers: whenever a customer completes the service, it is replaced by ν customers, where ν has a given branching distribution [7].

This single-server system is defined as follows:

- customers arrive according to a Poisson process with a rate λ;
- a newly arriving customer receives a type i with probability β_i, $(\beta_i \geq 0, \beta_1 + \ldots + \beta_n = 1)$;
- the service is nonpreemptive;
- the durations of service are independent random variables with the distribution function $B_i(t)(B_i(0) = 0)$ for i-type customers; suppose

$$b_i = \int_0^\infty t dB_i(t) < \infty, \quad b_{i2} = \int_0^\infty t^2 dB_i(t) < \infty; \tag{4}$$

- whenever the i-type customer completes service, it is replaced by ν_i customers, where $\nu_i = (k_1, k_2, \ldots, k_n)$ with probability $q_i(k_1, k_2, \ldots, k_n)$; by $Q_i(\mathbf{z}) = Q_i(z_1, z_2, \ldots z_n)$ denote a generating function of ν_i and suppose

$$q_{ij} = \frac{\partial Q_i}{\partial z_j}(1, \ldots, 1) < \infty, \qquad \frac{\partial^2 Q_i}{\partial z_j \partial z_k}(1, \ldots, 1) < \infty, \tag{5}$$

(because of the applications being modeled, our interest is confined to a system in which a branching process ν_i generates a total population with a finite expected size);
- new customers are immediately fed back to the ends of the corresponding queues;
- if the queue is not empty, the server immediately selects (by the service discipline) a customer and starts;
- by γ_i we denote the expected total amount of service provided to a customer of type i and all its population; suppose

$$\rho = \lambda \sum_{i \in \Omega} \beta_i \gamma_i < 1. \tag{6}$$

Let

$$L_i = \lim_{T \to \infty} \frac{1}{T} \int_0^\infty \mathbf{E} l_i(t) dt,$$

where $l_i(t)$ is the amount of i-type customers in the queue at a moment t $(1 \leq i \leq n)$.

Let $L_i(\mathbf{a}, \mathbf{b})$ be the mean length of customers of type i with the probabilistic-divided service discipline.

Our main theorem generalizes and improves the result of [8].

Theorem 1. *Let $L_1^*, \ldots, L_n^*$ be a mean length of queues under the stable regime with any service discipline, then there exist parameters $\mathbf{a}$ and $\mathbf{b}$ of the probabilistic-divided service discipline such that $L_i(\mathbf{a}, \mathbf{b}) = L_i^*$, $i = 1, 2, \ldots, n$.*

3 Proof of the Theorem

To prove this theorem, we need some notation.

Let λ_i be the rate of type i customers (arrive or appear after branching). Then

$$\lambda_i = \sum_{j=1}^n q_{ji} \lambda_j + \lambda \beta_i, \quad i = 1, 2, \ldots, n. \tag{7}$$

The proof is found in [7].

In paper [9], the second author has proved that

$$\begin{aligned} &\textstyle\sum_{i=1}^n L_i^* \gamma_i = \omega(\{1, 2, \ldots, n\}), \\ &\textstyle\sum_{i \in S} L_i^* \gamma_i(S) \geq \omega(S), \quad \forall S \subset \{1, 2, \ldots, n\}, \end{aligned} \tag{8}$$

where $\gamma_i(S)$ is the expected total amount of service provided to a customer of type $i \in S$ and all its population in S ($\gamma_i(\{1,2,\ldots,n\}) = \gamma_i$) and

$$\omega(S) = \inf_{a,b} \sum_{i \in S} L_i(a,b)\gamma_i(S).$$

Without loss of generality it can be assumed that

$$L_1^*/\lambda_1 \le L_2^*/\lambda_2 \le \ldots \le L_n^*/\lambda_n. \tag{9}$$

Under the conditions of the theorem and (9), we describe an algorithm for finding parameters **a** and **b** of the probabilistic-divided service discipline.

STEP 0. Let

$$\begin{aligned} a_1 &= 1; \\ a_{2i} &= i+1, 1 \le i \le n-1; \\ a_{2i+1} &= i, \qquad 1 \le i \le n-1; \\ a_{2n} &= n; \end{aligned}$$

$$\mathbf{b_0} = (0,0,1,0,1,0,1,0,1,0,\ldots,0,1,1),$$

and

$$L_i^0 = L_i(\mathbf{a}, \mathbf{b_0}), \quad i = 1,2,\ldots,n.$$

By S denote a subset of $\{1,2,\ldots,n\}$ such that

$$S = \{i : L_i^0 \ge L_i^*\}.$$

STEP 1. Solving the system of equations

$$L_i(\mathbf{a},\mathbf{b}) = (1-t)L_i^0 + tL_i^*, i = 1,2,\ldots,n, \tag{10}$$

with the complement condition

$$\text{if } j \in S \text{ then } b_j = 1 \text{ else } b_j = 0,$$

we get the solution $(t_j, \mathbf{d})$ for $j = 2,3,\ldots,n$.

Let

$$t_\alpha = \min_{2 \le j \le n} \{t_j\}.$$

If $t_\alpha = 1$, we find parameters **a** and **b** of the probabilistic-divided service discipline.

STEP 2. Let $\alpha \in S$ (therefore $d_\alpha = 0$).

Let

$$m = \min\{i : i > l(\alpha), a_i \in S, a_i > \alpha, l(a_i) = i\},$$
$$k = \max\{i : i > m, a_i \notin S, a_i < \alpha\}.$$

If max is not defined, we set $k = m$.

Displacing $a_{f(\alpha)} = \alpha$ after a_k, we get a new cortege

$$\mathbf{a} = (a_1, \ldots, a_{f(\alpha)-1}, a_{f(\alpha)+1}, \ldots, a_k, \alpha, a_{k+1}, \ldots, a_{2n}).$$

Go to STEP 1.

STEP 3. Let $\alpha \notin S$ (therefore $d_\alpha = 1$).

Let

$$m = \max\{i : i < f(\alpha), a_i \in S, a_i < \alpha, f(a_i) = i\},$$
$$k = \min\{i : i < m, a_i \notin S, a_i > \alpha\}.$$

If min is not defined, we set $k = m$.

Displacing $a_{l(\alpha)} = \alpha$ before a_k, we get a new cortege

$$\mathbf{a} = (a_1, \ldots, a_{k-1}, \alpha, a_k, \ldots, a_{l(\alpha)-1}, a_{l(\alpha)+1}, \ldots, a_{2n}).$$

Go to STEP 1.

The proof that such a new cortege always exists is found in [9].

4 Modification of the TCP

The main idea of the suggested TCP modification is the following: in the ARTCP (considered in [4], [5]) the round trip time interval is replaced by the length of the corresponding queue.

The experimental testing of the ARTCP, carried out in the paper [6], has shown its efficiency. At the same time we can apply theorem 1, which shows that on average our suggested modification of the TCP has the same posibilities as the ARTCP. Thus, our modification improves the TCP, but does not require the introduction of an additional field in the protocol.

Let us consider our TCP modification. Without entering in details of the ARTCP (see [4], [5]) we shall describe its main characteristic, namely, the packet transmission rate. It is exactly the characteristic we will modify. Other components of the ARTCP will be without changing.

Let us define the rate that is used in the ARTCP (see [4], [5]). For every sent packet (in the Round Trip Time (RTT) interval) the ARTCP memorizes the value of $\tau = t_1 - t_0$, where t_0 is equal to the time of the current packet sending off and t_1 is equal to the time of the next packet sending off. The receiver memorizes the value of τ in the corresponding field. Then the rate has the following value

$$R = S/\tau,$$

where S is the length of the corresponding packet.

In the suggested modification the value τ is replaced by the value l_i/λ_i, where l_i is the length of the queue from the packet set of the given type i and λ_i is defined in (7).

Instead of rate changing we change the corresponding parameter b_i.

The increment of the parameter b_i leads to the increment of the rate R.

If $b_i = 0$, then we change the parameters **a**, as described in STEP 2. The parameter **b** will be set up in its initial value.

If $b_i = 1$, then we change the parameters **a**, as described in STEP 3. The parameter **b** will be set up in its initial value.

5 Conclusion

Thus, the described construction allows us to elliminate the measurement in the service system. The application of this result to the TCP simplifies the ARTCP and doesn't require an additional field in the protocol. As a result, the use of probabilistic-divided discipline for organizing the work of the " eliminates some TCP disadvantages.

References

1. Stevens W.R. : TCP/IP Illustrated. Volume 1: The Protocols. Addison-Wesley, New York (1994)
2. Balakrishnan H., Padmanabhan V., Katz R. : The Effects of Asymmetry on TCP Performance. ACM MobiCom, **9** (1997)
3. Brakmo L., O'Malley S., Peterson L. : TCP Vegas: New Techniques for Congestion Detection and Avoidance. ACM SIGCOMM **8** (1994) 24-35
4. Alekseev I.V., Sokolov V.A. : The Adaptive Rate TCP. Model. and Anal. Inform. Syst. **6**, 1 (1999) 4–11
5. Alekseev I.V., Sokolov V.A. : Compensation Mechanism for Adaptive Rate TCP. First IEEE/Popov workshop on Internet Technologies and Services, (October 25-28, 1999), Proceedings. **2** (1999) 68-75
6. Alekseev I.V. : Model and Analysis Transmission Protocol ARTCP. Investigation in Russia **27** (2000) 395–404 http://zhurnal.ape.relarn.ru/articles/2000/027.pdf
7. Kitaev M.Yu., Rykov V.V. : A Service System with a Branching Flow of Secondary Customers. vtomatika i Telemechanika **9** (1980) 52–61
8. Timofeev E.A. : Probabilistic-divided service discipline and the polyphedron of mean waits in the system $GI|G|1$. vtomatika i Telemechanika **10** (1991) 121–125
9. Timofeev E.A. : Optimization of the mean lengths of queues in the service system with branching secondary flows of customers. vtomatika i Telemechanika **3** (1995) 60–67

From ADT to UML-Like Modelling

Short Abstract

Egidio Astesiano, Maura Cerioli, and Gianna Reggio

DISI-Universita' di Genova
Via Dodecaneso 35
16146 Genova, Italy
fax: +39-010-3536699, e-mail: astes@disi.unige.it

UML and similar modelling techniques are currently taking momentum as a de facto standard in the industrial practice of software development. As other Object Oriented modelling techniques, they have benefited from concepts introduced or explored in the field of Algebraic Development Techniques — for short ADT, formerly intended as Abstract Data Types — still an active area of research, as demonstrated by the CoFI project. However, undeniably, UML and ADT look dramatically different, even perhaps with a different rationale. We try to address a basic question: can we pick up and amalgamate the best of both? The answer turns out to be not straightforward. We analyze correlations,lessons and problems. Finally we provide suggestions for further mutual influence, at least in some directions.

Transformation of UML Specification to XTG

E. E. Roubtsova[1] and J. van Katwijk[2], R. C. M. de Rooij[2], W. J. Toetenel[2]

[1] Eindhoven University of Technology, Den Dolech 2, P.O.Box 513, 5600 MB The Netherlands
e-mail: E.Roubtsova@tue.nl
[2] TU Delft, Zuidplantsoen 4, NL-2628 BZ, The Netherlands
e-mail: W.J.Toetenel@its.tudelft.nl

1 Introduction

The Unified modeling language (UML) is the object modeling standard [1] that provides a set of diagrams for system specification from static and dynamic perspectives [6]. Nowadays, it is becoming more and more custom for designers to use formal methods during the design. The formal methods allow to verify a system specification with respect to its property specification. The system specification in UML can be formalized [4], however, the property specification is limited by the logic of the Object Constraint Language (OCL) [6] which does not allow to specify properties of computational paths, reachability etc. [2, 9].

This paper shows our approach to specification of systems and properties in UML. We consider UML class, object and statechart diagrams as an input language of the Prototype Model Checker (PMC) being under development in Delft University of technology [8]. This tool is intended to verify properties of real time systems represented in a variant of Time Computation Tree Logic (TCTL). We represent in this paper the transformation of the UML specification into the original input language of PMC, namely Extended Timed Graphs (XTG). The transformation allows to specify system properties by TCTL at the UML level and to verify them by means of PMC.

2 System Specification. Property Specification

1. *We have chosen three kinds of UML diagrams to specify a system.*
   - Class and object diagrams represent a static definition of the system. A class diagram specifies sets of classes Cl_S with their attributes At_S and operations Op_S. An object diagram defines a current set of class instances.
   - A UML statechart diagram addresses a dynamic view to the system. All labels of the statechart use names that are defined by the UML class and object diagrams [3].
2. *To enable the measurement of time* we introduce clock attributes of special *DenseTime* type. A clock attribute presents a timer that can be reset to a nonnegative real value. After resetting the value of the clock attribute continuously increases.
3. *A UML statechart is a tuple* $SchD = (S, S_0, Rl)$:
   - S is a tree of states. The states are depicted in the statechart diagrams by boxes with round corners. There are states of three types in this tree: *AND*,*XOR* and *Simple*. *AND* and *XOR* nodes are hierarchical states. One of them is a root of the tree of states. Nodes $A_1, \dots A_n$ of a hierarchical state C are drawn inside of C. An AND-state is divided by dotted lines to put other states between such lines. A simple state does not contain other states. In general, a state s is marked by $(Name_s, In_s, Out_s, History_s)$ labels. The labels $History_s$, In_s, Out_s can be empty.
   - $S_0 \subseteq S$ is a set of initial states.
   - Rl is a set of relations among states $Rl = \{TR, Conn, Synch\}$.
     - TR is a set of transitions that are represented by labeled arrows.

       $$TR = \{(s_i, s_j, e, g, a) | s_i, s_j \in S;\ e \in Op_S,\ g \in BooleanExpression(At_S, Op_S), a \in Op_S\}.$$

     - $Conn = \{(I, O) | I \subset S, O \subset S, \}$. The relation is represented by a black rectangle (fork-join connector) and by arrows that are directed from each state $I_i \in I$ to the connector and from the connector to each state $O_j \in O$.
     - $Synch = \{(I, O) | I \subset S, O \subset S\}$. The relation is drawn by two black rectangles (fork and join connectors) and by a circle of a synch-state.

A transition semantics of a UML-statechart, in general, was defined in [4] as a computational tree. A node of this tree is a vector $n = (s, v, q)$, where $s \subset S$ of the $SchD$, v is a tuple of values of all attributes of objects from the object diagram, q is a queue of operation calls from the $SchD$.

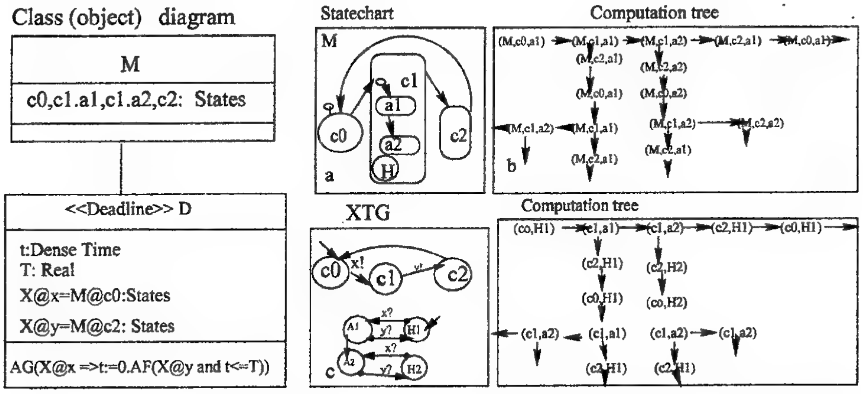


Fig. 1.

A simple example of a class M is shown in the fig.1. Assume that one object is statically defined. The statechart of the class has the XOR state c_1 with the $History$ mark H. The transition to the XOR state means the transition to the initial state a_1 among substates a_1, a_2. The transition from the XOR state c_1 means the transition from the current state a_i. Only one substate from the a-set can be the current substate. The history label means remembering of the current substate of the XOR state and starting the next computation inside of the XOR state from this substate. The semantics is shown by the computation tree (fig.1b).

5. *To enable the specification of properties* of computational sub-trees, we define specification classes. Specification classes are stereotyped. They can be related to a set of traditional classes. Each stereotype of specification has its own intuitive name and a formal representation by a parameterized TCTL formula [3]. The TCTL variant that we use [3,8] contains location predicates and reset quantifiers over variables. For example, there is an instance D of the *Deadline* stereotype in the class diagram (fig.1). The stereotype has the clock attribute t and the parameters: deadline T and two location predicates $X@x$ ("class X is in the state x"), $X@y$. Stereotype *Deadline* is defined by its predefined TCTL formula over parameters and attributes to specify timeliness in the system. The formula means "for every path always, if class X is in the state x and we reset clock $t := 0$, the state y of the class X is reachable at the moment when the value of clock t is less then the value of deadline T".

3 Transformation Approach

We transform a UML system specification into XTG, the original language of the Prototype Model Checker. XTG is a formalism for describing real-time systems. An XTG is a tuple $G = (V, L, l_0, T)$, *where*

- V is a finite set of variables of the following $DataType = \{Integer, Enumeration, Real, DenseTime\}$. $DenseTime$ is a type for representing clocks (as nonnegative real that increases continuously).
- L is a finite set of locations (nodes). $l_0 \in L$ is an initial location;
- T is a finite set of transitions (arrows with labels).
  $T = \{(l_i, l_j, c, up, ur)\}$, where $l_i, l_j \in L$, $c \in BooleanExpression(V)$, $up \in ValueAssignment$ of the variables V, $ur \in \{Urgent, UnUrgent\}$. An urgent transition is represented by an arrow with a black dot.

A state in the time computation tree semantics of the XTG is defined [8] by a location and the values of variables in the location (l, ρ). A transition from a state (l, ρ) is enabled if $c(\rho) = True$ after the substitution of the variable values. Time can be pass in state as long as its invariant is satisfied and no urgent transitions are enabled. Time is not allowed to progress while a transition marked as urgent is enabled. The parallel composition of XTG is defined [8] by a synchronization mechanism that is based on a form of value passing

CCS [5]. A synchronization is a pair of labels specifying that the transition marked by *syn*! in an XTG is executed simultaneously with a transition marked by *syn*? in another XTG.

We make the transformation of the UML specification into XTG in three steps.

First, we represent the hierarchical states from the statecharts introducing the CCS synchronization. The result of this transformation we name the flat statecharts. A flat statechart is a UML statechart which does not contain hierarchical states and uses CCS synchronization mechanism. The parallel composition of two flat statecharts is a flat statechart such that the set of its transitions is defined by rules of XTG parallel composition.

Second, we replace in the flat statecharts all operation calls using the synchronization.

The last step means the transformation of fork-join connectors and synch states of the flat statescharts to XTG.

In the complete paper we present all the transformation schemes. In this abstract the illustration by an example is given. We represent the statechart (fig.1) by two XTG: the external graph (c_0, c_1, c_2) and internal one (a_1, a_2). The history label means that there is a history state H_i for each a_i. States c_0, H_1 are initial for the system. The transition to the XOR state is replaced by two synchronous transitions: $(c_0, c_1, x!)$ and $(H_i, a_i, x?)$. The value of i depends on the last active state of the internal graph. The transition from the XOR state is replaced by pair of synchronous states $(c_1, c_2, y!)$ and $(a_i, H_i, y?)$.

The correctness of our transformation is shown by the computation tree of the XTG (fig.1d) which is equal to the computation tree of the statechart (fig.1b). For some of transformation schemes there is a projection function that maps the computation tree of the transformation result onto the computation tree of the transformation source.

The specification of properties given in UML is suitable for XTG. So, our approach allows to combine the PMC model checker power and the UML specification possibilities. The realization of the approach uses the extensibility interface [7] of the Rational Rose UML tool.

References

1. G. Booch, J. Rubaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, Amsterdam, 1999.
2. B. P. Douglass. *Real-Time UML. Developing Efficient Objects for Embedded Systems.* Addison-Wesley, 1998.
3. E.E.Roubtsova and J.van Katwijk and W.J.Toetenel and C.Pronk and R.C.M.de Rooij. The Specification of Real-Time Systems in UML. *MCTS2000,* http://www.elsevier.nl/locate/entcs/volume39.html, 2000.
4. J. Lilius and I.P.Palor. Formalising UML StateMachines for Model Checking. *UML'99. Beyond the Standard, LNCS 1723*, pages 430–445, 1999.
5. R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.
6. OMG. *Unified Modeling Language Specification v.1.3.* ad/99-06-10, http://www.rational.com/uml/resources/documentation/index.jsp, June 1999.
7. Rational Rose 98i. *Rose Extensibility Reference 2000.* http://www.rational.comwww.se.fh-heilbronn.de/usefulstuff/Rational Rose 98i Documentation.
8. R.F. Lutje Spelberg and W.J. Toetenel and M. Ammerlaan. Partition Refinement In Real-Time Model Checking. *Formal Techniques in Real-Time and Fault-Tolerant Systems. LNCS,* 1486:143–157, 1998.
9. B. Selic. Turning Clockwise: Using UML in the Real-Time Domain. *Communications of the ACM*, 42(10):339–355, October 1999.

Petri Nets

A Systematic Approach towards Object-Based Petri Net Formalisms

Berndt Farwer[1] and Irina Lomazova[2]

[1] University of Hamburg, Department of Computer Science, e-mail: farwer@informatik.uni-hamburg.de
[2] Program Systems Institute of Russian Academy of Science,
e-mail: irina@univ.botik.ru

Abstract. The paper aims at establishing the semantical background for extending Petri net formalisms with an object-oriented approach by bringing together Nested Petri Nets (NP-Nets) of Lomazova and Linear Logic Petri nets (LLPNs) of Farwer. An introductory example shows the capabilities of these formalisms and motivates the proposed inter-encoding of two-level NP-Nets and LLPNs. A conservative extension to the Linear Logic calculus of Girard is proposed – Distributed Linear Logic. This extension of Linear Logic gives a natural semantical background for multi-level arbitrary token nets.

1 Introduction

Modularisation and the object-oriented programming paradigm have proved to be facultative for the development of large-scale systems, for instance in flexible manufacturing, telecommunications, and workflow systems. These efficient approaches to systems engineering stimulated researchers in the Petri net community to develop new flavours of Petri nets, such as LOOPN of C. Lakos [6] and OPN of R. Valk [10], which incorporate object-oriented aspects into Petri net models (see also the survey [11], referring to a large collection of techniques and tools for combining Petri nets and object-oriented concepts). The design of these new net formalisms has led to the problem that only very few results from the traditional theory of Petri nets are preserved and thus the spirit of Petri net model is considerably altered. Moreover, many of the object-oriented extensions of Petri nets are made '*ad hoc* and lack a formal theoretical background.

The formalisms studied in the present paper take a more systematic view of objects in Petri nets in order to allow the application of standard Petri net techniques, and hence they do not follow the object-orientation known from programming languages in every aspect. Their study aims to give some insight into the way, how objects can be integrated into Petri nets not violating the basis of the initial model.

We consider two independently proposed extensions of ordinary Petri net formalisms, both of which utilize tokens representing dynamic objects. The idea of supplying net tokens with their own net structure and behaviour is due to R.Valk [10].

In Nested Petri nets (NP-Nets) [7] tokens are allowed to be nets themselves. In contrast to Object Petri nets of Valk, where an object net token may be in some sense distributed over a system net, in NP-Nets a net token is located in one place (w.r.t. a given marking). This property facilitates defining formal semantics for NP-Nets and makes possible a natural generalization of this model to multi-level and even recursive cases [8]. The main motivation for introducing NP-Nets was to define an object-oriented extension of Petri nets, which would have a clear and rigorous semantics, still be weaker than Turing machines and maintain such merits of Petri net models, as decidability of some crucial verification problems. It was stated in [9], that while reachability and boundedness are undecidable for NP-Nets, some other important problems (termination, coverability) remain decidable.

Linear Logic Petri nets (LLPNs) have been introduced as a means for giving purely logical semantics to object-based Petri net formalisms. The basic property that makes Linear Logic, introduced in 1989 by Girard

[5], especially well-suited for this task is the resource sensitivity of this logic . The nature of Linear Logic predestines it also for the specification of Petri nets with dynamic structure [2]. A Linear Logic Petri net is a high-level Petri net that has Linear logic formulae as its tokens. The token formulae can be restricted to a fragment of Linear Logic to maintain decidability. Arcs have multisets of variables as inscriptions and the transitions are guarded by sets of Linear Logic sequents that are required to be derivable in the applicable sequent calculus for the transition to occur. Then in [2] it was shown, that two-level NP-Nets also allow a natural Linear Logic representation, and thus have a very close relation to LLPNs.

Linear Logic of Girard has proved to be a natural semantic framework for ordinary Petri nets, so that the reachability of a certain marking in the net corresponds to the derivability of the associated sequent formula in a fragment $\mathbf{ILL}_{\mathbf{PN}}$ of the intuitionistic Linear Logic sequent calculus. In this paper we propose an extension of Linear Logic to Distributed Linear Logic, which allows, in particular, giving Linear Logic semantics to multi-level object nets. The main difference from classical Linear Logic will be in that resources, represented by formulae, will be distributed (belong to some owners or distributed in space), so that two resources located in different points (or belonging to different owners), generally speaking, cannot be used together. To express this we propose the use of the special binary operation P ("possess"). We write $A(\phi)$ for $P(A, \phi)$, where A is an atomic symbol, representing an owner or location, and ϕ is a Linear Logic formula (possibly including the "possess" operation).

There is a natural interpretation of "possess" operation, which continues Girard's example from [5] about possessing a dollar and buying a box of cigarettes. Let D be a dollar. Then $A(D)$ designates that A owns a dollar; $A(!(D \multimap C))$ means, that A has an ability to obtain cigarettes for a dollar (e.g. A is not a child). Further in this setting $A(D) \multimap B(D)$ means that A can pass his dollar to B, and $A(x) \multimap B(x)$ (universally quantified for x) means that A is ready to give B anything he has.

The paper is organized as follows. Section 2 contains a short example representing NP-Net and LLPN models. Section 3 is devoted to a formal translation of NP-Nets into LLPNs and vice versa and contains results presented in [3]. Here the NP-Nets are assumed to have only bounded element nets. In section 4 we introduce the extension of Linear Logic by the "possess" operator and use it for the Linear Logic representation of multi-level NP-Nets. Section 5 contains some conclusions.

2 Object-Based Modelling with NP-Nets and LLPNs

In NP-Nets tokens are nets. A behaviour of a NP-Net includes three kinds of steps. An autonomous step is a step in a net in some level, which may "move", "generate", or "remove" its elements (tokens), but doesn't change their inner states. Thus, in autonomous steps an inner structure of tokens is not taken into account. There are also two kinds of synchronization steps. Horizontal synchronization means simultaneous firing of two element nets, located in the same place of a system net. Vertical synchronization means simultaneous firing of a system net together with its elements "involved" in this firing. Formal definitions of NP-Nets can be found in [7, 8].

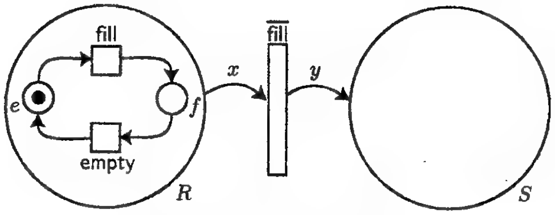
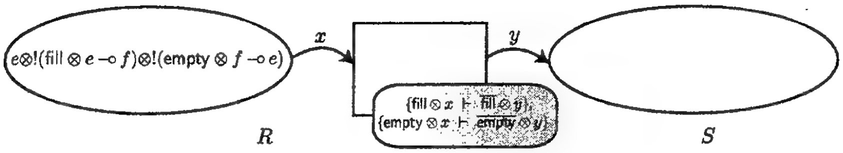


Fig. 1. Example NP-Net

Figure 1 shows an example of a NP-Net, where a system net has two places R and S and one transition marked by $\overline{\text{fill}}$. In the initial marking it has one element net (with two places e and f and two transitions marked by fill and empty) in R. Here only a vertical synchronization step via synchronization label fill is possible. It moves the element net token to S and simultaneously changes its inner marking to f.

An equivalent LLPN net is shown in Figure 2.

In LLPNs tokens are Linear Logic formulae that can be used to represent atoms or net components. Arcs are inscribed with (multisets of) variables. LLPNs have transition guards consisting of a set of Linear Logic



$e \otimes !(\text{fill} \otimes e \multimap f) \otimes !(\text{empty} \otimes f \multimap e)$
x
y
$\{\text{fill} \otimes x \vdash \overline{\text{fill}} \otimes y\},$
$\{\text{empty} \otimes x \vdash \overline{\text{empty}} \otimes y\}$
R
S

Fig. 2. LLPN equivalent to the NP-Net from Fig.1

sequents with the understanding that a transition may only occur if all sequents are derivable in the underlying calculus with the chosen binding of token formulae to the multiset of variables of the input and output arcs.

In addition, autonomous derivation steps can occur at any time, i.e. a formula α residing in some place may evolve to some other formula β without the occurrence of any transition in the Petri net, provided $\alpha \vdash \beta$ is provable in the underlying calculus.

A slight modification of the original definition given in [1] is used here. For a more compact representation the guard of the LLPN in Figure 2 uses a subset of the powerset of sequents with the intended meaning that the satisfaction of all sequents of one of these sets is a sufficient condition for the activation of the transition. In the original formalism two transitions would have to be used to represent the NP-Net's system net transition depicted in Figure 1.

The reader is referred to [1, 4] for an introduction to LLPNs. A discussion of further issues, especially dealing with dynamic modifications of object net structures can be found in [2].

3 Inter-Representability of Two-Level NP-Nets and LLPNs

This section gives a brief sketch of a formal translation of two-level NP-Nets into LLPNs and vice versa. It contains results presented in [3] and restricts the class of NP-Nets to those that have only bounded element nets. For a translation of multi-level NP-Nets refer to the extension of the Linear Logic calculus proposed in section 4.

The main issue of any translation of an object Petri net formalism into a LLPN framework is how to deal with synchronization. Instead of giving the complete translation of a given NP-Net into its corresponding LLPN (which can be found in [3]), we sketch the main idea of synchronization within LLPNs. The formal translation then comes as an obvious consequence.

For each pair of system and element net transitions t_s and t_e with adjacent vertical synchronization labels l and $\bar{l}$ we define two new unique propositional symbols with the same name as the labels. W.l.o.g. assume these labels to be disjoint with any place names used in either net. We call the pair of labels *message handles* for synchronization of t_s and t_e.

Let, for example, $!(A \multimap B)$ be a partial canonical formula of an element net $\mathcal{N}$ residing in place p of a Linear Logic Petri net $\mathcal{LLPN}$. The synchronization of the system net transition t_s with a transition t_e in the element net via message handles $(l, \bar{l})$ is represented by the token formula

$$!(l \otimes A \multimap \bar{l} \otimes B)$$

and by a transition in $\mathcal{LLPN}$ with the guard function

$$G(t) = \{l \otimes x \ \vdash \ \bar{l} \otimes y\}.$$

Here the message handles are used to ensure that the derivation step, which takes place during the firing of the system net transition of the LLPN, really uses the Linear Logic implication representing the element net transition t_e.

Thus, a synchronization of a firing in the system net with the derivation step outlined above coincides with the vertical synchronization step of the simulated NP-Net. Horizontal synchronization can be simulated analogously.

The simulation relies on the boundedness of the NP-Net, since it uses a propositional calculus for the encoding. A generalization of the calculus to overcome this restriction is discussed in the remainder of the paper.

Theorem 1. *For every unary elementary NP-Net NPN without constants in arc inscriptions, there exists a canonical Linear Logic Petri net $\mathcal{LLPN}_{NPN}$, such that*

- *the token formula in $\mathcal{LLPN}_{NPN}$ is the canonical representation of the element net of NPN,*
- *there is a bijection between the transitions of NPN and $\mathcal{LLPN}_{NPN}$, which generates the bijecton between occurrence sequences of the system net in NPN and occurrence sequences of $\mathcal{LLPN}_{NPN}$.*

On the contrary, the simulation of LLPNs by NP-Nets is possible only for a fairly restricted class of LLPN, since the definition of LLPNs is much more general. It allows variables to occur multiply in input and output arc inscriptions and the use of transition guards. [3] gives some conditions for the possibility of a NP-Net simulation for LLPNs.

4 Extending the Linear Logic Calculus

Linear Logic of Girard due to its resource sensitivity has a close resemblance to Petri nets: it has connectives that can handle resources in the same manner as ordinary Petri nets do. A Linear Logic representation of high-level Petri nets, such as e.g. coloured Petri nets of K. Jensen, can be achieved by simulating the "unfolding" of places and transitions (cf. [1]), when for each token colour a new copy of the place is created (technically, the set of places indexed by the possible colours represents the new set of propositional atom symbols for the encoding).

To obtain a straightforward Linear Logic encoding of high-level Petri nets the intuitionistic Linear Logic calculus $\mathbf{ILL}_{\mathrm{PN}}$, used so far to encode Petri nets, can be extended from propositional to predicate level. Now formulae may contain variables, and all variables are supposed to be universally quantified. Then we introduce a special possess operation P for encoding that a place A contains a token x. Note, that components of NP-Nets are high-level nets with net tokens, so the formula of the form $P(A, \phi)$ will designate, that a net token with encoding ϕ belongs to the place A.

More formally, we extend the language $L(\mathbf{ILL}_{\mathrm{PN}})$ of intuitionistic Linear Logic to the language $L(\mathbf{DILL}_{\mathrm{PN}})$ by adding the binary operation $P(\cdot,\cdot) \subseteq \mathcal{A} \times L(\mathbf{DILL}_{\mathrm{PN}})$, where $\mathcal{A}$ is a set of atomic symbols, representing owners or locations, in the following way.

On the syntactical level we have:

- a finite set of atom symbols $A, B, \ldots$ (corresponding to places);
- a finite set of atom symbols $a, b, \ldots$ (corresponding to token colours);
- variables $x, y, \ldots$;
- the binary logical operations $\otimes$, $\multimap$;
- the modality "of course" !;
- the binary operation symbol P (for possess).

Formulae (terms) are defined as follows:

- token atoms $a, b, \ldots$ and variables $x, y, \ldots$ are terms;
- if A is a place atom, ϕ — a term, then $P(A, \phi)$ is a term. We write $A(\phi)$ as a shorthand for $P(A, \phi)$;
- if ϕ and ψ are terms, then $\phi \otimes \psi$, $\phi \multimap \psi$ and $!\phi$ are terms.

As usual for sequent calculi we define a sequent formula as an expression of the form $\Gamma \vdash \phi$, where Γ is a sequence of terms and ϕ is a term.

Variables are interpreted as terms. The $\mathbf{DILL}_{\mathrm{PN}}$ calculus is obtained by adding to $\mathbf{ILL}_{\mathrm{PN}}$ the following rule for substituting a term for a variable:

$$\frac{\Gamma \vdash \phi}{\Gamma[\psi/x] \vdash \phi[\psi/x]}$$

Now we come to defining a Linear Logic encoding of a NP-Net by canonical formula. Here in the Linear Logic encoding of a current marking a separate factor of the form $A(\phi)$, where ϕ is an encoding of the corresponding net token, will represent a token occurrence in the place A. Analogously to the LLPN representation of NP-Nets, given in the previous section, new propositional symbols with the same names as labels are used for encoding horizontal and vertical synchronizations.

Definition 1 (canonical formula for NP-Net). *The* extended canonical formula *of a NP-Net $\langle P, T, W, \mathbf{m}\rangle$ is defined by the tensor product of the following factors. W.l.o.g., assume that each arc is carrying either a variable, or a constant:*

- *For each autonomous transition $t \in T$ construct the factor*

$$!\left(\bigotimes_{p\in {}^\bullet t} p(W(p,t)) \multimap \bigotimes_{q\in t^\bullet} q(W(t,q))\right),$$

- *For each transition $t \in T$ with a (vertical or horizontal) synchronization label l, construct the factor*

$$!\left(l \otimes \bigotimes_{p\in {}^\bullet t} p(W(p,t)) \multimap \bar{l} \otimes \bigotimes_{q\in t^\bullet} q(W(t,q))\right),$$

- *For each transition $t \in T$ with a vertical synchronization label l, construct the factor*

$$!\left(\left(\bigotimes_{x\in\{x_1,\ldots,x_k\}} (\bar{l} \otimes x \multimap l \otimes x') \otimes \bigotimes_{p\in {}^\bullet t} p(W(p,t))\right) \multimap \right.$$
$$\left. \bigotimes_{q\in t^\bullet} q(W(t,q))[x'_1/x_1]\ldots[x'_k/x_k]\right),$$

  where $x_1,\ldots,x_k$ are all variables occurring in input arc expressions of t.
- *For each place $p \in P$ and a horizontal synchronization label $\lambda \in L$, construct the factor*

$$!\left(((\bar{\lambda} \otimes x) \multimap (\lambda \otimes x') \otimes (\lambda \otimes y) \multimap (\bar{\lambda} \otimes y') \otimes p(x) \otimes p(y)) \multimap (p(x') \otimes p(y'))\right)$$

- *For each element token a and a vertical synchronization label $l \in L$, construct the factor*

$$!\left((l \otimes a) \multimap (\bar{l} \otimes a)\right),$$

- *Construct for the current marking* $\mathbf{m}$ *and all places $p \in P$ and tokens a with $a \in \mathbf{m}(p)$ the formulae $p(a)$. So, for the complete marking we get*

$$\bigotimes_{\substack{p\in P\\ a\in\mathbf{m}(p)}} p(\Psi_{\mathbf{DILL_{PN}}}(a)),$$

  where multiple occurrences of tokens contribute to multiple occurrences of factors in the tensor product.

Thus, for example, the NP-Net shown in Figure 1 is translated into the formula:
$R(e\otimes!(\mathsf{fill} \otimes e \multimap f)\otimes!(\mathsf{empty} \otimes f \multimap e))\otimes!((\mathsf{fill} \otimes x \multimap \overline{\mathsf{fill}} \otimes x') \otimes R(x) \multimap S(x'))$

In this example we do not need factors corresponding to vertical synchronization of element tokens with inner ones, for it's a two-level net.

Theorem 2. *The sequent formula*

$$\Psi_{\mathbf{DILL_{PN}}}(\langle\mathcal{N},\mathbf{m}\rangle) \ \vdash \ \Psi_{\mathbf{DILL_{PN}}}(\langle\mathcal{N},\mathbf{m}'\rangle),$$

where $\Psi_{\mathbf{DILL_{PN}}}(\langle\mathcal{N},\mathbf{m}\rangle)$ is a canonical formula for a marked NP-Net $\langle\mathcal{N},\mathbf{m}\rangle$, is derivable in $\mathbf{DILL}_{\text{PN}}$ *iff* $\mathbf{m}'$ *is reachable from $\langle\mathcal{N},\mathbf{m}\rangle$.*

5 Conclusion

The main focus of our work on object-oriented extensions of Petri nets has been to establish core formalisms that have a clear mathematical semantics and preserve some important decidability results (see [7, 8]) that are lost by many *ad hoc* extensions of the basic Petri net formalism. The independent definition of the two similar formalisms of Linear Logic Petri nets and nested Petri nets suggest that the foundations of these formalisms are sound and have a strong theoretical background.

On the other hand, extending Linear Logic to deal with multi-level nested Petri nets leads to the logical framework, which has a natural interpretation not only for Petri nets, but for many applications, where a possibility of using these or other resources depends on their owners and/or locations.

References

1. Farwer B. *A Linear Logic View of Object Petri Nets.* Fundamenta Informaticae, 37:225–246, 1999.
2. Farwer B. *A Multi-Region Linear Logic Based Calculus for Dynamic Petri Net Structure.* Fundamenta Informaticae, 43:61–79, 2000.
3. Farwer B. *Relating Formalisms for Non-Object-Oriented Object Petri Nets.* In Proceedings of the Concurrency Specification and Programming (CS&P'2000) Workshop. 9–11 October 2000. Vol. 1, pp. 53–64. Informatik-Bericht Nr.140, Humboldt-Universität zu Berlin, Informatik-Berichte, Berlin, 2000.
4. Farwer B. *Linear Logic Based Calculi for Object Petri Nets.* Dissertation, Universität Hamburg. Published by: Logos Verlag, ISBN 3-89722-539-5, Berlin, 2000.
5. Girard J.-Y. *Linear Logic.* Theoretical Computer Science, 50:1–102, 1987.
6. Lakos C. A. *From Coloured Petri Nets to Object Petri Nets.* Proc. Int. Conf. on Appl. and Theory of Petri Nets, pp. 278–297. LNCS 935. Springer-Verlag, 1995.
7. Lomazova I. A. *Nested Petri Nets — A Formalism for Specification and Verification of Multi-Agent Distributed Systems.* Fundamenta Informaticae, 43:195–214, 2000.
8. Lomazova I. A. *Nested Petri Nets: Multi Level and Recursive Systems.* In Proceedings of the Concurrency Specification and Programming (CS&P'2000) Workshop. 9–11 October 2000. Vol. 1., pp. 117–128. Informatik-Bericht Nr.140, Humboldt-Universität zu Berlin, Informatik-Berichte, Berlin, 2000.
9. Lomazova I. A. and Schnoebelen Ph. *Some Decidability Results for Nested Petri Nets.* In: Proc. Andrei Ershov 3rd Int. Conf. Perspectives of System Informatics (PSI'99), Novosibirsk, Russia, July 1999, pp. 207–219. LNCS 1755, Springer-Verlag, 1999.
10. Valk R. *Petri Nets as Token Objects: An Introduction to Elementary Object Nets.* In: Proc. Int. Conf. on Application and Theory of Petri Nets, LNCS 1420, Springer-Verlag, pp. 1–25, 1998.
11. Zapf M., Heinzl A. *Techniques for Integrating Petri Nets and Object-Oriented Concepts.* Working Papers in Information Systems, No.1/1998. University of Bayreuth, 1998.

Unfoldings of Coloured Petri Nets

Vitaly E. Kozura

A. P. Ershov Institute of Informatics Systems
Siberian Division of the Russian Academy of Sciences
6, Acad. Lavrentiev ave., 630090, Novosibirsk, Russia
e-mail: vkozura@iis.nsk.su

Abstract. In this paper the unfolding technique is applied to coloured Petri nets (CPN) [6,7]. The technique is formally described, the definition of a branching process of CPN is given. The existence of the maximal branching process and the important properties of CPN's unfoldings are proven. A new approach consisting in combining unfolding technique with symmetry and equivalence specifications [7] is presented and the important properties of resulting unfoldings are proven. We require CPN to be finite, n-safe and containing only finite sets of colours.

1 Introduction

The state space exploring in Petri net (PN) analysis is one of the most important approaches. Unfortunately, it faces the state explosion problem. Among the approaches which are used to avoid this problem are the stubborn set method, symbolic binary decision diagrams (BDD), methods based on partial orders, methods using symmetry and equivalence properties of the state space, etc. [14].

In [12] McMillan proposed an unfolding technique for PN analysis. In his works, instead of the reachability graph, a finite prefix of maximal branching process, large enough to describe a system, has been considered. The size of unfolding is exponential in the general case and there are few works which improve in some way the unfolding definitions and the algorithms of unfolding construction [5,8].

Initially McMillan has proposed his method for the reachability and deadlock analysis (which has also been improved in the later work [11]). J.Esparza has proposed a model-checking approach to unfolding of 1-safe systems analysis [3]. In [1] the unfolding technique has been applied to timed PN. In [2,4] LTL-based model-checking on PN's unfolding has been developed. Unfolding of coloured Petri nets has been considered in the general case in [13] for using it in the dependence analysis needed by the Stubborn Set method.

In the present paper the application of the unfolding method based on earlier works for ordinary PNs to coloured Petri nets (CPN) [6,7] is given. This allows to construct the finite unfolding for CPN and apply the reachability and deadlock analysis methods for it. It allows to consider applying the model-checking technique to unfoldings of CPN, as well. The technique is formally described, the definition of a branching process of CPN is given. The existence of the maximal branching process and the important properties of CPN's unfoldings are proven.

In [7] symmetry and equivalence specifications for CPN are introduced. In the present paper it is also presented a new approach consisting in combining unfolding technique with symmetry and equivalence specifications. This allows an additional reduction of the size of CPN's unfolding.

2 Coloured Petri Nets

In this section we briefly remind the basic definitions related to coloured Petri nets and describe the subclass of colours we will use in the paper. More detailed description of CPN can be found in [6].

A *multi-set* is a function m: $S \to N$, where S is a usual set and N is the set of natural numbers. In the natural way we can define operations such as $m_1 + m_2$, $n \cdot m$, $m_1 - m_2$, and relations $m_1 \leq m_2$, $m_1 < m_2$. Also $|m|$ can be defined as $|m| = \sum_{s \in S} m(s)$. Let $Var(E)$ define the set of variables of the expression E, and $Type(E)$ define the type of the expression E.

A *coloured Petri net (CPN)* is the net N $= (S, P, T, A, N, C, G, E, I)$, S, P, T, A are the sets of colours, places, transitions, and arcs such that $P \cap T = P \cap A = T \cap A = \emptyset$, N is a mapping $N : A \to (P \times T) \cup (T \times P)$, C is a colour function $C : P \to S$, G is a guard function such that for all $t \in T$ $Type(G(t)) = bool$ and $Type(Var(G(t))) \subseteq S$, E is the function defined on arcs with $Type(E(a)) = C(p)_{MS}$, where p is the place from $N(a)$ and $Type(Var(E(a))) \subseteq S$ and I is the initial function defined on places, such that for all $p \in P$ $Type(I(p)) = C(p)_{MS}$.

$A(t), Var(t), A(x,y), E(x,y)$ can be defined in the natural way.

A *binding* b is a function from $Var(t)$ such that $b(v)\in Type(v)$ and $G(t)\langle b\rangle$. The set of bindings for t will be denoted by $B(t)$. A *token element* is a pair (p, c) where $p\in P$ and $c\in C(p)$. The set of all token elements is denoted by TE. A *binding element* is a pair (t, b) where $t\in T$ and $b\in B(t)$. The set of all binding elements is denoted by BE. A *marking* M is a multi-set over TE. A *step* Y is a multi-set over BE. A step Y is *enabled* in the marking M if for all $p\in P$ $\sum_{(t,b)\in Y} E(p,t)\langle b\rangle \leq M(p)$ and a new marking M_1 is given by $M_1(p) = M(p) - \sum_{(t,b)\in Y} E(p,t)\langle b\rangle + \sum_{(t,b)\in Y} E(t,p)\langle b\rangle$.

Now we can define a subclass of coloured Petri nets, which is large enough to describe many interesting systems and still allows us to build a finite prefix of its branching process. The detailed description can be found in [9]. The set of basic colour domains is obtained from the types of *Standard ML (SML)* [6] by allowing to consider only finite colour domains $s\in S$. All functions defined in [6] and having the above described classes as their domains are allowed in our subclass. The CPN satisfying all the above-mentioned requirements is called *S-finite.*

The marking M of a CPN is *n-safe* if $|M(p)|\leq n$ for all $p\in P$. A CPN is called *n-safe* if all of its reachable markings are n-safe. 1-safe net is also called *safe.* A *preset* of an element $x\in P\cup T$ denoted by ${}^\bullet x$ is the set ${}^\bullet x = \{y\in P\cup T \mid \exists a : N(a) = (y, x)\}$. A *postset* of x denoted by $x^\bullet$ is the set $x^\bullet = \{y\in P\cup T \mid \exists a : N(a) = (x, y)\}$.

The CPN considered in this paper are the CPN satisfying three additional properties:

1. The number of places and transitions is finite.
2. The CPN is n-safe.
3. The CPN is S-finite.

3 Branching Process of Coloured Petri Nets

Let $\mathbf{N}$ be a Petri net. We will use the term *nodes* for both places and transitions. The nodes x_1 and x_2 are *in conflict*, denoted by $x_1\sharp x_2$, if there exist transitions t_1 and t_2 such that ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$ and (t_1,x_1) and (t_2,x_2) belong to the transitive closure of N (which we denote by R_t). The node x is in *self-conflict* if $x\sharp x$. We will write $x_1\leq x_2$ if $(x_1, x_2)\in R_t$ and $x_1 < x_2$ if $x_1\leq x_2$ and $x_1\neq x_2$. We say that x *co* y, or $x||y$, or x *concurrent* y if neither $x < y$ nor $x > y$ nor $x\sharp y$.

An Occurrence Petri Net (OPN) is an ordinary Petri net $\mathbf{N} = (P, T, N)$, where

1. P, T are the sets of places and transitions,
2. $N\subseteq(P\times T)\cup(T\times P)$ gives us the incidence function,

satisfying the following properties:

1. For all $p\in P$ $|p|\leq 1$,
2. N is acyclic, i.e., the (irreflexive) transitive closure of N is a partial order.
3. N is finitely preceded, i.e. for all $x\in P\cup T$ the set $\{y\in P\cup T \mid y\leq x\}$ is finite which gives us the existence of $Min(\mathbf{N})$, the set of minimal elements of $\mathbf{N}$ with respect to R_t.
4. no transition is in self conflict.

Let $\mathbf{N}_1 = (P_1, T_1, N_1)$ and $\mathbf{N}_2 = (P_2, T_2, N_2)$ be two Petri nets. A *homomorphism* h from $\mathbf{N}_2$ to $\mathbf{N}_1$ is a mapping $h : P_2\cup T_2 \rightarrow P_1\cup T_1$ such that

1. $h(P_2)\subseteq P_1$ and $h(T_2)\subseteq T_1$.
2. for all $t\in T_2$ $h|_{{}^\bullet t} = {}^\bullet t \rightarrow {}^\bullet h(t)$.
   for all $t\in T_2$ $h|_{t^\bullet} = t^\bullet \rightarrow h(t)^\bullet$.

Now we give the main definition of the section. This is the first novelty of the paper, a formal definition of a branching process for coloured Petri nets. After the following definition, the existence result is given.

Definition 1 *A branching process of a CPN* $\mathbf{N}_1 = (S_1, P_1, T_1, A_1, N_1, C_1, G_1, E_1, I_1)$ *is a tuple* $(\mathbf{N}_2, h, \varphi, \eta)$, *where* $\mathbf{N}_2 = (P_2, T_2, N_2)$ *is an OPN, h is a homomorphism from* $\mathbf{N}_2$ *to* $\mathbf{N}_1$, φ *and* η *are the functions from* P_2 *and* T_2, *respectively, such that*

1. $\varphi(p)\in C_1(h(p))$.
2. $\eta(t)\in B(h(t))$.
Other requirements are listed below:
3. for all $p_1\in P_1$ $\sum_{p\in Min(\mathbf{N}_2)\ |\ h(p)=p_1}\varphi(p) = M_0(p_1)$,
4. $G_1(h(t))\langle\eta(t)\rangle$ *for all* $t\in T_2$.

5. $\forall t'\in T_2 \mid (\exists a\in A_1 : N_1(a) = (p,t) \textit{ and } h(t') = t) \Longrightarrow$
   $E_1(a)\langle\eta(t')\rangle = \sum_{(p'\in{}^{\bullet}t' \ \mid \ h(p')=p)}\varphi(p')$,
   $\forall t'\in T_2 \mid (\exists a\in A_1 : N_1(a) = (t,p) \textit{ and } h(t') = t) \Longrightarrow$
   $E_1(a)\langle\eta(t')\rangle = \sum_{(p'\in t'^{\bullet} \ \mid \ h(p')=p)}\varphi(p')$,
6. *If* $(h(t_1) = h(t_2))$ *and* $(\eta(t_1) = \eta(t_2))$ *and* $({}^{\bullet}t_1 = {}^{\bullet}t_2)$ *then* $t_1 = t_2$.

Using the first two properties, we can associate a token element (p,c) of $\mathbf{N}_1$ with every place in $\mathbf{N}_2$ and the binding element (t,b) of $\mathbf{N}_1$ with every transition in $\mathbf{N}_2$. So we can further consider the net $\mathbf{N}_2$ as containing the places which we identify with token elements of $\mathbf{N}_1$, and transitions which we identify with binding elements of $\mathbf{N}_1$. So we sometimes use them instead, like $h((t,b)) = t$ means $h(t') = t$ and $\eta(t') = b$ or $p\in{}^{\bullet}(t,b)$ means $p\in{}^{\bullet}t'$ and $h(t') = t$ and $\eta(t') = b$. Analogously, we can consider $(p,c)\in P_2$ as $p'\in P_2$ and $h(p') = p$ and $\varphi(p) = c$. Also, $h(p,c) = p$ and $h(t,b) = t$.

It can be shown that any finite CPN has a maximal branching process (MBP) up to isomorphism (theorem 1). We can declare existence of the maximal branching process when considering the algorithm of its generation. The algorithm is described in [9] and the following theorem is proven there.

Theorem 1 *For a given CPN* $\mathbf{N}$ *there exists a maximal branching process MBP(*$\mathbf{N}$*)*

This branching process can be infinite even for the finite nets if they are not acyclic. We are interested in finding a finite prefix of a branching process large enough to represent all the reachable markings of the initial CPN. This finite prefix will be called an unfolding of the initial CPN.

4 Unfoldings of CPN

A *configuration* C of an OPN $\mathbf{N} = (P,T,N)$ is a set of transitions such that $t\in C \Longrightarrow$ for all $t_0 \leq t$, where $t_0\in C$ and for all $t_1,t_2\in C$ $\neg(t_1\sharp t_2)$. A set $X_0\subseteq X$ of nodes is called a *co-set*, if for all $t_1,t_2\in X_0$: $(t_1 \text{ co } t_2)$. A set $X_0\subseteq X$ of nodes is called a *cut*, if it is a maximal co-set with respect to the set inclusion.

Finite configurations and cuts are closely related. Let C be a finite configuration of an occurrence net, then $Cut(C) = (Min(\mathbf{N})\cup C^{\bullet})^{\bullet}C$ is a cut.

Let $\mathbf{N}_1 = (S_1,P_1,T_1,A_1,N_1,C_1,G_1,E_1,I_1)$ be a CPN and MBP$(\mathbf{N}_1) = (\mathbf{N}_2,h,\varphi,\eta)$, where $\mathbf{N}_2 = (P_2,T_2,N_2)$, be its maximal branching process. Let C be a configuration of $\mathbf{N}_2$. We define a marking *Mark(C)* which is a marking of $\mathbf{N}_1$ such that $Mark(C)(p) = \sum_{(p'\in Cut(C) \ \mid \ h(p')=p)} M_2(p')$.

Let $\mathbf{N}$ be an OPN. For all $t\in T$ the configuration $[t] = \{t'\in T \mid t'\leq t\}$ is called a *local configuration.* (The fact that [t] is a configuration can be easily checked).

Let us consider the maximal branching process for a given CPN. It can be noticed that MBP(N) satisfies the completeness property, i.e., for every reachable marking M of $\mathbf{N}$ there exists a configuration C of MBP(N) (i.e., C is the configuration of OPN) such that $Mark(C) = M$. Otherwise we could add a necessary path and generate a larger branching process. This would be a contradiction with the maximality of MBP(N).

Now we are ready to define three types of cutoffs used in the definition of unfolding. The first two definitions for ordinary PNs can be found in [3, 12]. The last is the definition given in [8].

Definition 2 *Let* $\mathbf{N}$ *be a coloured Petri net and MBP(*$\mathbf{N}$*) be its maximal branching process. Then*

1. *A transition* $t\in T$ *of an OPN is a* GT_0*-cutoff, if there exists* $t_0\in T$ *such that* $Mark([t]) = Mark([t_0])$ *and* $[t_0]\subset[t]$.
2. *A transition* $t\in T$ *of an OPN is a GT-cutoff, if there exists* $t_0\in T$ *such that* $Mark([t]) = Mark([t_0])$ *and* $\|[t_0]\| < \|[t]\|$.
3. *A transition* $t\in T$ *of an OPN is a EQ-cutoff, if there exists* $t_0\in T$ *such that*
   (a) $Mark([t]) = Mark([t_0])$
   (b) $\|[t_0]\| = \|[t]\|$
   (c) $\neg(t\|t_0)$
   (d) *there are no EQ-cutoffs among t' such that* $t'\|t_0$ *and* $\|[t']\|\leq\|[t_0]\|$.

Definition 3 *For a coloured Petri net* $\mathbf{N}$*, an unfolding is obtained from the maximal branching process by removing all the transitions t', such that there exists a cutoff t and* $t < t'$*, and all the places* $p\in t'^{\bullet}$*. If Cutoff =* GT_0*(GT)-cutoffs, then the resulted unfolding is called* GT_0*(GT)-unfolding.* GT_0*(GT)-unfolding is also called the McMillan unfolding. If Cutoff = GT-cutoffs ∪ EQ-cutoff, then the resulted unfolding is called EQ-unfolding.*

It has been shown that the McMillan unfoldings are inefficient in some cases. The resulting finite prefix grows exponentially, when the minimal finite prefix has only a linear growth. The following proposition can be formulated for these three types of unfoldings ([9]).

Proposition 1 *EQ-unfolding $\leq$ GT-unfolding $\leq$ GT_0-unfolding.*

The following theorem presents the main result of this section ([9]).

Theorem 2 *Let* **N** *be a CPN. Then for its unfoldings we have:*

1. *EQ-unfolding, GT-unfolding and GT_0-unfolding are finite.*
2. *EQ-unfolding, GT-unfolding and GT_0-unfolding are safe, i.e., if C and C' are configurations, then $C \subseteq C' \implies Mark(C') \in [Mark(C)\rangle$.*
3. *EQ-unfolding, GT-unfolding and GT_0-unfolding are complete, i.e., $M \in [M_0\rangle \implies$ there exists a configuration C such that Mark(C) = M.*

In the general case the algorithm proposed in [12] and applied to coloured Petri nets in [9] has an exponential complexity. The algorithm from [8] is rather efficient in the speed of unfolding generation. In the case of an ordinary PN it gives the overall complexity $O(N_P \cdot N_T)$, where N_P and N_T are the numbers of places and transitions in EQ-unfolding. This algorithm was also transferred to coloured Petri nets [9] and a close estimation holds if we don't take into consideration the calculation complexity of arc and guard functions. In this case we obtain $O(N_P \cdot N_T \cdot B)$, where $B = max\{|B(t)| \ : \ t \in T_{CPN}\}$.

5 Unfoldings with Symmetry and Equivalence

In this part the technique of equivalence and symmetry specifications for coloured Petri nets (CPN) will be applied to the unfolding nets of CPN. It will be shown how to generate the maximal branching process and its finite prefixes for a given CPN under the equivalence or symmetry specifications. All symmetry and equivalence specifications are taken from [6].

Let **N** be a CPN and **M** and BE be the sets of all markings and binding elements of **N**. The pair $(\approx_M, \approx_{BE})$ is called an *equivalence specification* if $\approx_M$ is an equivalence on **M** and $\approx_{BE}$ is an equivalence on BE. $M_\approx$ and $BE_\approx$ are the equivalence classes. We say $(b, M) \approx (b^*, M^*)$ iff $b \approx_{BE} b^*$ and $M \approx_M M^*$. Let us have $X \subseteq \mathbf{M}$ and $Y \subseteq M_\approx$, then we can define: $[X] = \{M \in \mathbf{M} \mid \exists x \in X : M \approx_M x\}$ — the set of all markings equivalent to the markings from X and $[Y] = \{M \in \mathbf{M} \mid \exists y \in Y : M \in y\}$ — the set of all markings from the classes from Y. The equivalence specification is called *consistent* if for all $M_1, M_2 \in [[M_0\rangle]$ we have $M_1 \approx_M M_2 \implies [Next(M_1)] = [Next(M_2)]$, where $Next(M_1) = \{(b, M) \in BE \times \mathbf{M} | \ M_1[b\rangle M\}$.

A *symmetry specification* for a CP-net is a set of functions $\Phi \subseteq [\mathbf{M} \cup BE \rightarrow \ \mathbf{M} \cup BE]$ such that $(\Phi, \bullet)$ is an algebraic group and $\forall \phi \in \Phi : \phi|_\mathbf{M} \in [\mathbf{M} \rightarrow \mathbf{M}]$ and $\phi|_{BE} \in [BE \rightarrow BE]$. Each element of Φ is called a *symmetry*. A symmetry specification Φ is it consistent iff the following properties are satisfied for all symmetries $\phi \in \Phi$, all markings $M_1, M_2 \in [M_0\rangle$ and all binding elements $b \in BE$ $\phi(M_0) = M_0$ and $M_1[b\rangle M_2 \Longleftrightarrow \phi(M_1)[\phi(b)\rangle \phi(M_2)$.

Now the cutoff criteria will be defined for a CPN with a symmetry specification Φ or equivalence specification ϕ. We call the finite prefix of the maximal branching process of CPN obtained by using new cutoff criteria an *unfolding with symmetry* Unf^Φ or unfolding with equivalence $Unf^\approx$. Since accordingly to described above we can consider the symmetry specification as the case of equivalence specifications, we give the cutoff definitions only for equivalence specifications.

Taking into consideration the consistency of the regarded equivalence, we can conclude that it is sufficient to consider the classes [M] in our definitions of cutoffs. The classes of binding elements will be obtained in a natural way.

Definition 4 *Let* **N** *be a coloured Petri net and MBP(***N***) be its maximal branching process. Then*

1. *A transition $t \in T$ of an OPN is a $GT_0{}^{\approx_M}$-cutoff if there exists $t_0 \in T$ such that $Mark([t]) \approx_M Mark([t_0])$ and $[t_0] \subset [t]$.*
2. *A transition $t \in T$ of an OPN is a $GT^{\approx_M}$-cutoff if there exists $t_0 \in T$ such that $Mark([t]) \approx_M Mark([t_0])$ and $||[t_0]|| < ||[t]||$.*
3. *A transition $t \in T$ of an OPN is a $EQ^{\approx_M}$-cutoff if there exists $t_0 \in T$ such that*
   (a) $Mark([t]) \approx_M Mark([t_0])$
   (b) $||[t_0]|| = ||[t]||$
   (c) $\neg(t || t_0)$
   (d) *there are no EQ-cutoffs among t' such that $t' || t_0$ and $||[t']|| \leq ||[t_0]||$.*

The notion $Unf^\approx$ is used for any type of unfoldings.

Proposition 2 *$EQ^{\approx}$-unfolding $\leq GT^{\approx}$-unfolding $\leq GT_0{}^{\approx}$-unfolding.*

The following theorem presents the main result of this section [9].

Theorem 3 *Let* N *be a CPN and $\approx = (\approx_M, \approx_{BE})$ be a consistent equivalence on* N*. Then for an $Unf^{\approx}$*(N) *we have:*

1. *$[M] \in [[M_0]\rangle \iff \exists C$, a configuration of $Unf^{\approx}$*(N) *$\mid Mark(C) \approx_M M$.*
2. *$C \subset C'$ and C' is a configuration of $Unf^{\approx}$*(N) *$\iff [Mark(C')] \in [[Mark(C)]\rangle$.*

6 Net Example

As an example let us consider the CPN representing the producer-consumer system [7] (see Appendix). We consider the case when the buffer capacity $nb = 1$. As an equivalence specification, the abstraction from the data d_1 and d_2 is considered. The table 1 represents the results.

Let us notice that we should generate EQ-unfolding when using the symmetry specification. In the case of equivalence specifications in general we can use all types of unfoldings.

7 Conclusion

In this paper the unfolding technique proposed by McMillan in [12] and developed in later works is applied to coloured Petri nets as they are described in [6, 7]. The technique is formally described, the definition of a branching process of CPN is given. The existence of the maximal branching process and the important properties of CPN's unfoldings are proven. All the necessary details are presented in [9].

The unfolding is a finite prefix of the maximal branching process. To truncate the occurrence net, we consider three cutoff criteria in the paper. To construct the finite prefix, two algorithms of unfolding generation were formally transferred from the ordinary PN's area [9]. The complexities of these algorithms are discussed in this paper.

One of the important novelties of the paper is the application of the unfolding technique to CPN with symmetry and equivalence specifications as they are represented in [7]. The size of unfolding is often much smaller than the size of the reachability graph of a PN. Using the symmetry and equivalence specifications in the unfolding generation, we can additionally reduce the size of CPN's unfolding.

We require a CPN to be finite, n-safe and to contain only finite sets of colours.

In the future it is planned to construct finite unfoldings of Timed CPN as they are described in [6], using the technique of unfolding with equivalence (the first results are already obtained in [10]), and also to make all the necessary experiments with unfoldings of coloured Petri nets including the implementation of the model-checking method.

Acknowledgments. I would like to thank Valery Nepomniaschy for drawing my attention to this problem and Elena Bozhenkova for valuable remarks.

References

1. **B.Bieber, H.Fleischhack H:** Model Checking of Time Petri Nets Based on Partial Order Semantics. Proc. CONCUR'99. — Berlin a.o.: Springer-Verlag, 210-225 (1999).
2. **J.-M.Couvreur, S.Grivet, D.Poitrenaud:** Designing an LTL Model-Checker Based on Unfolding Graphs. Lecture Notes in Computer Science, Vol.1825, 123-145 (2000).
3. **J.Esparza:** Model-Checking Using Net Unfoldings. Lecture Notes in Computer Science Vol.668, 613-628 (1993).
4. **J.Esparza J, K.Heljanko:** A New Unfolding Approach to LTL Model-Checking. Lecture Notes in Computer Science Vol.1853, 475-486 (2000).
5. **J.Esparza, S.Romer, W.Vogler:** An Improvement of McMillan's Unfolding Algorithm Proc. TACAS'96. — Berlin a.o.: Springer-Verlag, pp. 87-106 (1997).
6. **K.Jensen:** Coloured Petri Nets. Vol. 1. Berlin a.o.: Springer, (1995).
7. **K.Jensen:** Coloured Petri Nets. Vol. 2. Berlin a.o.: Springer, (1995).
8. **A.Kondratyev, M.Kishinevsky, A.Taubin, S.Ten:** A Structural Approach for the Analysis of Petri Nets by Reduced Unfoldings. 17th Intern. Conf. on Application and Theory of Petri Nets, Osaka, June 1996. Berlin a.o.: Springer-Verlag, 346-365 (1996).
9. **V.E.Kozura:** Unfoldings of Coloured Petri Nets. Tech. Rep. N 80, Novosibirsk 34 pages (2000).
10. **V.E.Kozura:** Unfoldings of Timed Coloured Petri Nets. Tech. Rep. N 82, Novosibirsk 33 pages (2001).

11. **S.Melzer, S.Romer:** Deadlock Checking Using Net Unfoldings. Proc. of the Conf. on Computer Aided Verification (CAV'97), Haifa, pp. 352-363 (1997).
12. **K.L.McMillan:** Using Unfolding to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits. Lecture Notes in Computer Science Vol.663, 164-174 (1992).
13. **A.Valmari:** Stubborn Sets of Coloured Petri Nets. Proc. of the 12th Intern. Conf. On Application and Theory of Petri Nets. — Gjern, 102-121 (1991).
14. **A.Valmari:** The State Explosion Problem. Lecture Notes in Computer Science Vol.1491, 429-528 (1998).

Appendix

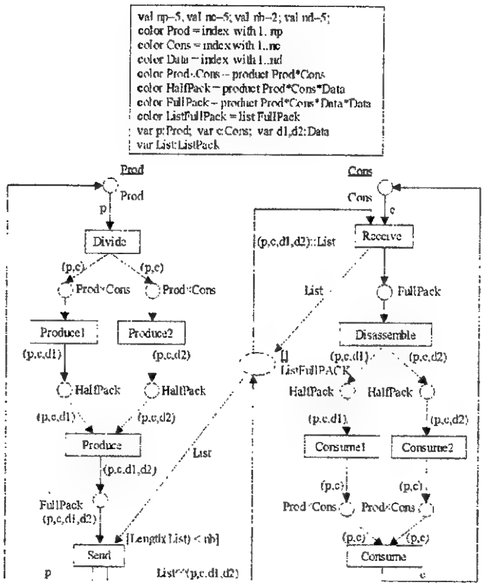


Fig. 1. Producer-Consumer system

Let us calculate here the sizes of reachability graphs and unfoldings for the producer-consumer system.

The number of reachable markings is

$N = (1 + c + 2{\cdot}c{\cdot}d + 2{\cdot}c{\cdot}d^2)^p{\cdot}(1 + p + 2{\cdot}p{\cdot}d + 2{\cdot}p{\cdot}d^2)^c{\cdot}(1 + p{\cdot}c{\cdot}d^2)$.

The unfolding consists of four parts (we denote it by PA, PB, PC and CA). When a producer initially produces data, the part PA is working. Part PB may work after a producer laid the first data to the buffer, but a consumer still cannot begin his part. Finally, PC is the part when a consumer definitely begins his work and a producer fulfills the buffer again. A consumer has the unique part CA. We have $|PA| = |PB| = |CA| = 5$ and $|PC| = 4$. The whole size is 19. When adding either one more producer or one more consumer, we come to the situation of doubling of $|PA|$, $|PB - 1|$ and $|CA|$ and adding the square of the number of parts $|PC + 1|$. Adding one more data acts as adding the square number of possibilities. Finally the size of the unfolding is

$$UnfSize = |PA|{\cdot}np{\cdot}nc{\cdot}nd^2 + |CA|{\cdot}np{\cdot}nc{\cdot}nd^2 + \\ |PB - 1|{\cdot}np{\cdot}nc{\cdot}nd^2 + |PC + 1|{\cdot}(np{\cdot}nc{\cdot}nd^2)^2.$$

As an equivalence specification, the abstraction from the data d_1 and d_2 is considered. For a graph this means that we can put nd=1. In the case of unfolding we obtain the additional (d-1) transitions in the part PA. The whole size of EQ-unfolding with this equivalence is

$UnfSize^{\approx} = |PA|{\cdot}np{\cdot}nc + |CA|{\cdot}np{\cdot}nc + |PB - 1|{\cdot}np{\cdot}nc + |PC + 1|{\cdot}(np{\cdot}nc)^2 + (d - 1)$. The table bellow represents the results.

p	c	d	O-graph	Consistent OE-graph	Unfolding's Size	EQ-unfolding with equivalence
1	1	1	72	72	19	19
3	3	3	$1.58{\cdot}10^{13}$	$1.68{\cdot}10^{8}$	$3.3{\cdot}10^{4}$	533
10	10	5	$1.32{\cdot}10^{69}$	$1.43{\cdot}10^{36}$	$3.1{\cdot}10^{7}$	$5.14{\cdot}10^{4}$
10	10	10	$7.8{\cdot}10^{70}$	$1.43{\cdot}10^{36}$	$5.0{\cdot}10^{8}$	$5.14{\cdot}10^{4}$
20	20	20	$1.73{\cdot}10^{174}$	$5.97{\cdot}10^{82}$	$1.28{\cdot}10^{11}$	$8.0{\cdot}10^{5}$
50	50	20	$2.11{\cdot}10^{469}$	$2.32{\cdot}10^{243}$	$5.0{\cdot}10^{12}$	$3.1{\cdot}10^{7}$

Table 1

A Net-Based Multi-Tier Behavior Inheritance Modeling Method*

Shengyuan Wang, Jian Yu, Chongyi Yuan

Department of Computer Science and Technology
Peking University, Beijing, 100871, China
e-mail: Wwssyy@263.net, yuj@theory.cs.pku.edu.cn, Lwyuan@pku.edu.cn

Abstract. A multi-tier methodology is required in order to make a smooth transformation from one stage to the next in the course of software development under a consistent conceptual framework. We present, in this paper, a multi-tier behavior inheritance modeling method based on Petri Nets, or to be precise, based on STLEN and DCOPN that are two net models serving as the tools for describing behaviors at two consecutive modeling tiers respectively.

Key Words: Concurrency, Object Orientation, Petri Net, Behavior Inheritance, Modeling Method.

The integration of Petri Nets with object orientation techniques has become promising ([1]–[8]). Parallelism, concurrency and synchronization are easy to model in terms of Petri Nets, and many techniques and software tools are already available for Petri Net analysis. These advantages have made Petri nets quite suitable to model the dynamic behavior of concurrent objects.

A concurrent object-oriented system consists of a dynamically varying configuration of concurrent objects operating in parallel. For different stages in the software development of such sort of systems, diversified Petri Net models may be found in the literature to perform the specification and/or verification of system behavior respecting that stage. Some net models are suitable to be used during the early stages ([5], [7], [8]), and others during the later ones ([1], [3], [4], [6]). Up to now, however, there is a lack of net-based formal methods that will guarantee a smooth transformation from one stage to the next under a consistent conceptual framework. This has prevented net-based methods from being more widely applied to the modeling of a concurrent object-oriented system, since incremental developing is one of the principles in object-oriented methodology. So we are persistent on the opinion that multi-tier methodology is necessary. Each tier is a significant specification phase in which at least one net-based model may be chosen as the modeling language. A multi-tier method should guarantee that the system behavior specified in one tier to be preserved in its successor tier, though the specification in the successor tier is usually more detailed.

A practical multi-tier method has to take into account all the primary elements of concurrent object-orientation, such as object (class) representation, inheritance, aggregation, cooperation (association), concurrency (intra/inter objects), etc. In this paper, we present a multi-tier inheritance modeling method, and two Petri Net models, belonging to two tiers respectively, are proposed to illustrate this method. The net model in the super-tier is a modified EN-system, called STLEN, in which both S-elements and T-elements are labeled. And in the successor tier (or sub-tier) is a net model called DCOPN (*dynamically configured object net*), which has s flavor of concurrent object-oriented programming languages, like net models in [1], [3], [4].

Formally, A *ST-Labelled EN system*, abbreviated as STLEN system, is a tuple $S = (N, \beta)$, where

(1)$N = (B, E; F, c_{in})$ is a Elementary Net system [14]. B and E are the set of *S-elements* and the set of *T-elements* respectively, $F \subseteq S \times T \cup T \times S$ is the *flow relation*, and $c_{in} \subseteq B$ is the *initial case.*

(2)$\beta : B \cup E \to L \cup \{\lambda\}$ is a labeling function such that

$$\forall b \in B, \forall e \in E.[\beta(b) \neq \lambda \vee \beta(e) \neq \lambda \to b(b) \neq b(e)].$$

(3) $\lambda \notin \beta(c_{in})$.

Where L is the set of identifiers that range over a name space, and $\lambda \notin L$ denotes the *unobservable* S-elements or T-elements. For $x \in B \cup E$, x is *observable* iff $\beta(x) \neq \lambda$. β is generally not an injection. Besides the unobservable elements being labeled with λ, several observable S-elements may be labeled with a single name (identifier), and the same is true for T-elements. From the definition above, we have $(\beta(B) - \{\lambda\}) \cap (\beta(E) - \{\lambda\}) = \Phi$.

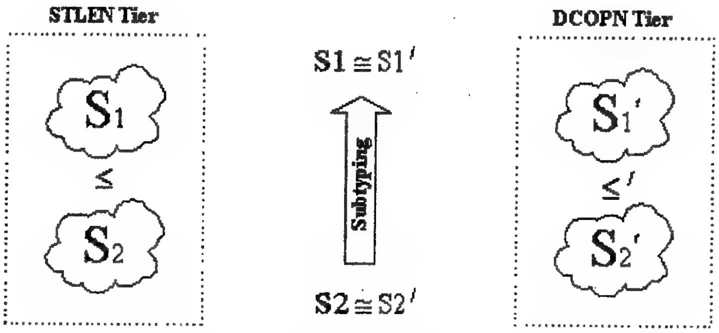
For the definition of dynamic behaviors of a STLEN system, one may refer to [9].

* This work was Supported by the National Natural Science Foundation of China under grant No. 69973003, and by the China NKBRSF (973) under grant G1999032706.

The formal definition of DCOPN is a little tedious (the details can be found in [10]). We will describe both STLEN and DCOPN informally with examples in our full paper.

Let DCOPN be the subsequent net model of STLEN in our multi-tier modeling method. For the behavior preserving from a STLEN tier net to its corresponding DCOPN one, a *restricted bisimulation relation*, denoted by $\cong$, between them has been defined. The word *restricted* conforms to the incremental design process from the STLEN tier to the DCOPN tier, i.e., the specification in the later is more detailed, or more restricted. Furthermore, the definition of the relation is based on the grouping of S-elements.

The term *inheritance* has often twofold meanings in the literature: the "code" reuse and the behavior preserving. In many places and also in this paper, the later is the meaning for another term *subtyping*, and the term inheritance stands simply for the former. In the situation of sequential object orientation, we do not usually distinguish between inheritance and subtyping, but it is very helpful to emphasize the distinction between them in the issues of concurrent object orientation, for example, in the comprehension of *inheritance anomaly*[12].



$$S2 \leq S1 \wedge S1 \cong S1' \wedge S2 \cong S2' \wedge S2' < S1' \Rightarrow S2' \leq' S1'$$

Fig. 1.

In the net-based multi-tier behavior inheritance modeling method of this paper, the subtyping relations are supposed to be preserved under the behavior preserving relations between two consecutive tier net models. We illustrate this in Figure 1 by the two tier situation from the STLEN tier to the DCOPN tier, in which the relation $\leq$ can be preserved to the relation $\leq'$ under the restricted bisimulation relation $\cong$, where $\leq$ is the subtyping relation between STLCN nets, and $\leq'$ is that between DCOPN nets. Each S_i represents a STLEN net, and each S_i' a DCOPN net. The relation $<$ represents the least subtyping relation between two DCOPN Nets, which makes the interface of a subtype net usable in any context in which the interface of one of its supertype nets can be used. To satisfy $<$ is a prerequisite in the definition of $\leq'$, which is the requirement to conform to the Principle of Substitutability [13]:*An instance of a subtype can always be used in any context in which an instance of a supertype was expected.*

One of the choices for the definition of $\leq$ is the one in [9], which is a bisimulation based on the grouping of S-elements and in terms of blocking or encapsulating actions, i.e., the external actions special to the subtype objects are to be inhibited in the context of the supertype objects.

In the definition of $\leq'$ in [10], a bisimulation relation is established between the sets of *attribute predicates* of two DCOPN nets, also in terms of blocking actions. The set of attribute predicates serves for describing the observable state of a DCOPN net , which is used in the definition of the relation $\cong$ too. Besides, to satisfy $<$ is a prerequisite as stated above.

For the definitions of $\leq$, $\leq'$, $\cong$ and $<$ in [10], we can prove the proposition showed in Figure 1.

Subtyping is a behavior preserving relation. Instead, inheritance is used for the code/specification reuse. Practically, to implement the behavior preserving while the code/specification is also highly reused, some incremental inheritance relation paradigms, capable of implementing the anticipant subtyping relations, need to be developed [8], the more the better. In our multi-tier inheritance modeling method, incremental inheritance relations are required to be preserved from the super-tier to the sub-tier, i.e., the THEN part in Figure 2 holds. One of the possibilities to obtain this is to ensure the IF part in Figure 2 to be satisfied.

The following is a guide for the behavior inheritance modeling method in this paper:

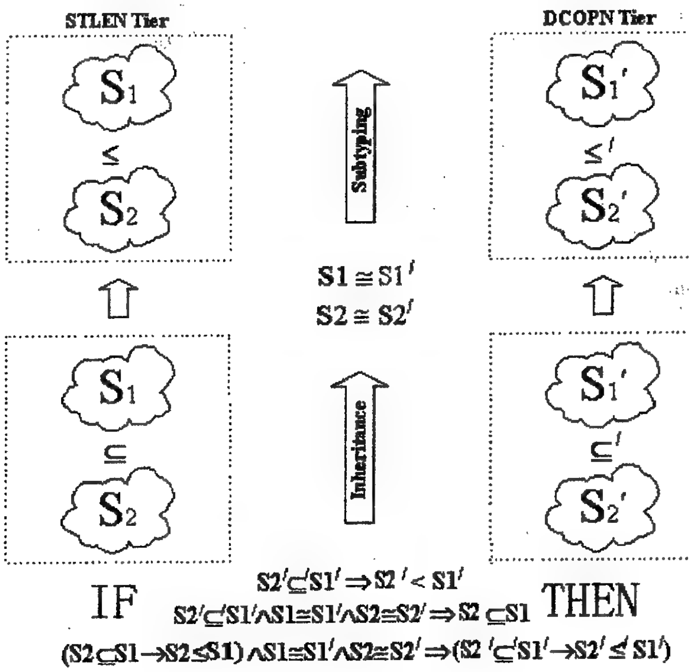

STLEN Tier
DCOPN Tier
S_1
$\leq$
S_2
S_1'
$\leq'$
S_2'
Subtyping
$S1 \cong S1'$
$S2 \cong S2'$
Inheritance
S_1
$\subseteq$
S_2
S_1'
$\subseteq'$
S_2'
IF
$S2' \subseteq' S1' \Rightarrow S2' < S1'$
$S2' \subseteq' S1' \wedge S1 \cong S1' \wedge S2 \cong S2' \Rightarrow S2 \subseteq S1$
THEN
$(S2 \subseteq S1 \rightarrow S2 \leq S1) \wedge S1 \cong S1' \wedge S2 \cong S2' \Rightarrow (S2' \subseteq' S1' \rightarrow S2' \leq' S1')$

Fig. 2.

(1) With the help of any OOA/OOD methodology, develop the behavior model of objects/classes using the net language STLEN and its available tools (to be developed). Label both the states (places) and actions (transitions), and the same time divide the states into groups according to the attributes.

(2) Develop DCOPN nets from STLEN ones by adding details, such as data types, constants, and attribute predicates. Build a map between the STLEN tier and the DCOPN tier in the same time when a DCOPN net is developed from its corresponding STLEN one. The map will be used in the verification of the restricted bisimulation relation $\cong$.

(3) Complete the interface specification for each DCOPN net. Verify the restricted bisimulation relations between them.

(4) For the behaviour inheritance (subtyping) modeling, just consider the derived net in the STLEN tier first. Then develop the DCOPN net from the derived STLEN one according to (2) and (3). Don't forget that the interface specifications for each super DCOPN class net and its derived DCOPN class net, developed from STLEN one, have to satisfy the least subtyping relation $<$.

(5) Develop and use incremental inheritance paradigms as many as possible. This may substantively save the work, as illustrated in Figure 2.

(6) Changes in the modeling process are allowable, which is simplified in our method since the direct corrections in the DCOPN tier may be avoided.

Many aspects for the multi-tier methodology still need to be explored. Researchers who are interested in it may find many new topics about it.

References

1. Charles Lakos. From Coloured Petri Nets to Object Petri Nets. Proceedings of 16th Int. Conference on the Application and Theory of Petri Nets, LNCS 935, Turin, Italy, Springer-Verlag,1995.
2. E.Batiston, A,Chizzoni, Fiorella De Cindo. Inheritance and Concurrency in CLOWN. Proceedings of the "Application and Theory of Petri Net 1995" workshop on "Object-oriented programs and models concurrency", Torino, Italy 1995.
3. C.Sibertin-Blanc. Cooperative Nets. Proceedings of 15th Int. Conference on the Application and Theory of Petri Nets. LNCS 815, Zaragoza, Spain, Springer-Verlag, 1994.
4. D.Buchs and N.Guelfi. CO-OPN: A Concurrent Object Oriented Petri Net Approach. Proceedings of 12th Int. Conference on the Application and Theory of Petri Nets, LNCS 524, Gjern, Denmark, 1991.

5. R.Valk. Petri Nets as Token Objects—An Introduction to Elementary Object Nets. Proceedings of 19th Int. Conference on the Application and Theory of Petri Nets, LNCS 1420, Springer-Verlag, 1998.
6. U.Becker and D.Moldt. Object-oriented Concepts for Coloured Petri Nets. Proceedings of IEEE Int. Conference on System, Man and Cybernetics, vol. 3, 1993, pp 279-286.
7. A.Newman, S.M.Shatz, and X.Xie. An Approach to Object System Modeling by State-Based Object Petri Nets. Journal of Circuits, Systems and Computers, Vol.8, No.1(1998) 1-20.
8. W.M.P. Vander Aalst and J.Basten. Life-Cycle Inheritance: A Petri-Net-Based Approach. 18th Int. Conference on the Application and Theory of Petri Nets, LNCS1248, Toulouse, France, Springer-Verlag, 1997.
9. S.Y.Wang, J.Yu and C.Y.Yuan. A Pragmatic Behavior Subtyping Relation Based on Both States and Actions. To appear in Journal of Computer Science and Technology, vol.16, 2001.
10. S.Y.Wang. A Petri Net Modeling Method for Concurrent Object-Oriented Systems. PhD thesis, Peking University, June 2001.
11. S.Christensen, N.D.Hansen. Coloured Petri Nets Extended with Place Capacities, Test Arcs and Inhibitor Arcs. Proceedings of 14th Int. Conference on the Application and Theory of Petri Nets, LNCS 691, Chicago, USA, Springer-Verlag, 1993.
12. Satoshi Matsuoka and Akinori Yonezawa. Analysis of Inheritance Anomaly in Object-oriented Concurrent Programming Languages. In Reserch Directions in Concurrent Object-oriented Programming, edited by G.Agha, P.Wegner and A.Yonezawa, The MIT Press, pp.107-150, 1993.
13. P.Wegner,S.B.Zdonik. Inheritance as an Incremental Modification Mechanism or What Like is and Isn't Like. In ECOOP'88 Proceedings. Lecture Notes in Computer Science 322, pp 55-77, Springer-Verlag, 1988.
14. C.Y.Yuan: Principles of Petri Nets, Electronic Industry Press, Beijing, China, 1998.

Testing

Specification Based Testing: Towards Practice*

A. K. Petrenko

Institute for System Programming of Russian Academy of Sciences (ISPRAS),
B. Communisticheskaya, 25, Moscow, Russia
e-mail: petrenko@ispras.ru, Web: http://www.ispras.ru/~RedVerst/

Abstract. Specification based testing facilities are gradually becoming software production aids. The paper shortly considers current state of the art, original ISPRAS/RedVerst experience, and outlines the ways for further research and testing tool development. Both conceptual and technical problems of novel specification based testing technologies introduction are considered.

1 Introduction

The specification based testing (SBT) progressively moves from academic research area into real-life practice. The process of learning to handle SBT techniques has to overcome a lot of problem related to both technical and human/management facets of software development. Below we focus on technical problems, namely, on issues of specification and test suite development or, more specifically, we try to answer:

- **Why limited use** — why SBT has not been widely introduced in industry practice yet?
- **What is the best** specification approach[1]?
- **Which feature first** — which SBT features should be provided first?

The work is mainly grounded on the practical experience of RedVerst[2] group of ISPRAS [27]. The experience was gained from industrial projects under contracts with Nortel Networks (http://www.nortelnetworks.com), Advanced Technical Services APS and research projects under grants of RFBR (http://www.rfbr.ru) and Microsoft Research (http://www.research.microsoft.com).

2 State of the SBT Practice — Why Limited Use?

State of the SBT art is very dynamic. During last 5–6 year a lot of sound results have been produced in research and industry spheres. The attention of academic researchers in formal specification is being shifted from analytical verification to problems of test generation from formal specification. The considerable number of testing tool producers have announced features related to SBT. The most progress has been achieved in specific areas like telecommunication protocol testing. There are successful attempts to deploy formal specification and

* The work was partially supported by RFBR grant 99-01-00207.

[1] We mainly consider the common Application Program Interface (API) testing and basically we do not focus on specific specification and testing methods intended for a specific kind of software like telecommunication, compilers, databases, and so on.

[2] RedVerst stands for Research and development for Verification, specification and testing

SBT features for verification and validation of wide spectrum of software including API testing. But most commercial tool/technologies provide features for only partial specification (like assertions that describe the only part of functionality). The technologies do not provide instructive methodologies for specification and test design. Therefore the deployment of the technologies faces troubles in scalability of the approach in real-life projects. Other important problems are the integration of SBT tools and techniques with widely used Software Development Environment (SDE) and the introduction of the new activities in the conventional SoftWare Development Processes (SWDP). No one SBT tool does provide a complete set of features that meet common requirements of specification and test designers. Instead these tools try to suggest a "best and unique solution". ADL/ADL2 story is a sad example of such approach. The ADL/ADL2 family of specification notations provided quite powerful and flexible features for specification of C, C++, and Java interfaces functionality. But test design problems (both methodological and tool support ones), especially in context of OO testing, were neglected. It seems this reason has caused the refuse ADL use in industrial practice.

As promised we restrict our consideration with only technical issues. However, exhaustive above heading question answer has to involve, in addition, human and management facets (see more detail consideration in [15,19]).

3 Specification Approaches — What is the Best?

There are a few kinds of classification of specification approaches like model-oriented vs. property-oriented and state-based vs. action-based. To shortly review advantages and drawbacks of the specification approaches we will hold to following classification: *executable, algebraic* (usually, co-algebraic), *use cases or scenarios, and constraint specifications* (some specification kinds like temporal, reactive, etc. are outside of our consideration because their specifics).

Executable specifications, executable models. This approach implies developing a prototype to demonstrate feasibility and functionality of further implementation. The examples of the approach are SDL [20], VDM [19,23,26], explicit function definitions in RAISE [21]. Finite State Machines (FSM) and Petri nets could be considered as (more abstract) executable specifications too.

Algebraic specification provides a description of properties of some operations compositions (serial, parallel, random, etc.). Usually this approach is tightly related to axiomatic approach [1,9]. SDL follows this approach to specify data types [1,20]; RAISE [21] provides quite powerful facilities for axiomatic specification.

Use case/Scenario based specification approach suggests considering the scenarios of use instead of properties of the implementation. The approach is developed and propagated by OMG/UML community [28]. SDL community uses MSC [14] notation for scenario description. The informative review of the scenario-based testing techniques is presented in [19].

Constraint specification implies a description of data type invariants and pre- and post-conditions for each operation (function, procedure). There are specific techniques for OO classes and objects specification. The constraint specification approach is followed by VDM [3,19], Design-by-contract in Eiffel [24], implicit function definition style in RAISE, iContract [10], ADL [22].

From testing perspective the advantages and the drawbacks of the specification approaches could be evaluated by simplicity of the specification development and simplicity of the test derivation from the specification. For example, algebraic and FSM like specifications are very suitable for the test sequence generation including case of concurrent and distributed software. However, the approach provides very restricted opportunities in test oracle[3] generation, so real-life software designers face some troubles in attempt specifying their software using only algebraic or FSM approach. Besides the algebraic specification is non-scalable approach. Such specifications for toy example are very short and attractive, however as the size increases, the understandability of the algebraic specification drastically drops (however there exists a society of experts in algebraic specification who do not share this opinion).

So, in short, the heading question answer is "there is no the best specification approach". Specification and test designers need good composition of specification techniques. One example of such composition is a com-

[3] *Test oracle* is a decision procedure that automatically compares actual behavior of a target program (outcome of a target operation) against its specification [17].

bination of constraint and FSM specification used in [7,12]. Other example is SDL/MSC/TTCN composition that is widely used in SDL users society.

4 RedVerst Experience and Lessons Learned

Before answering third above declared question "Which feature first?" lets dwell on the lessons learned from the RedVerst experience. ISPRAS organized the RedVerst group accepting the challenge of Nortel Networks in 1994. The original goal of the group was developing a methodology applicable to conformance testing of API of Nortel Networks proprietary real-time operating system. By the end of 1996 RedVerst has developed the KVEST methodology [7,8,16,25], the specifications and the tools for test generation and test execution. The RAISE Specification Language (RSL) [21] was used for specification. The KVEST included techniques for automatic and semi-automatic test generation, automatic test execution and test result analysis and reporting. The techniques were oriented onto use in real-life processes, so, some practical requirements must been met like: fault tolerant testing, fully automatic re-generation, re-run of the tests and test result analysis. The total size of KVEST formal specifications now is over 200 Kline. 6 patent applications have been filed based on the KVEST research and development experience, a few patents have been taken out.

The most valuable KVEST solutions were as follows:

- A few kinds of test scenario schemes. The simplest scheme was intended for separate testing pure functions (without side effect) and allowed fully automatic test generation. The most complex schemes allow testing parallel execution of software like resource managers and messaging systems.
- Enhanced test generation technique that allows excluding from consideration the inaccessible and redundant test situations.
- Programming language independent technology scheme for test generation.
- Automatic integration of generated and manually developed components of test suites for semi-automatic test generation. The technique allows to exclude any manual customization during repeated re-generations of test suites. The feature is valuable for both test design and regression testing periods.

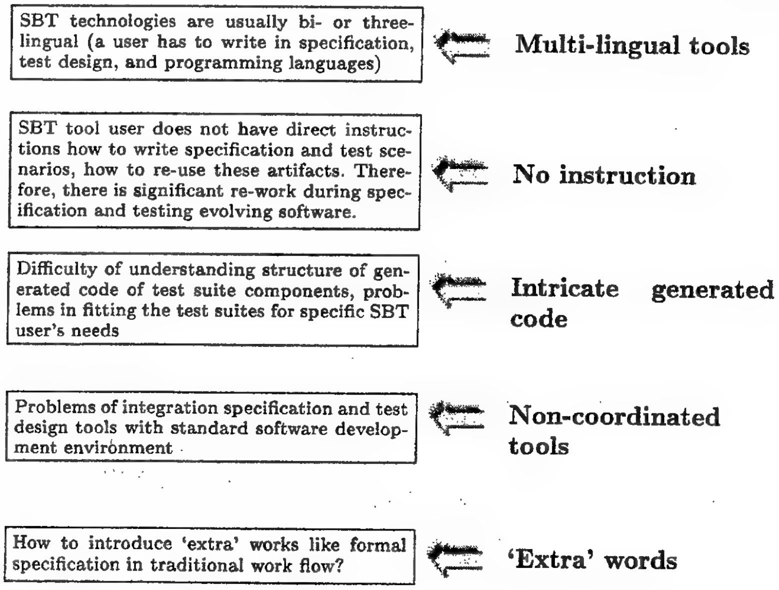
Up to now KVEST users have gained successful experience in verification of the following kinds of software.

- Operating system kernel and utilities
- Fast queuing systems for multiprocessor systems and for ATM framework
- Telecommunication protocols as a whole and some protocol implementation subsystems like protocol parsers.

Software verification processes. The KVEST has been applied in two kinds of software verification processes. First one is "Legacy reverse-engineering and improving process" and second one is "Regression testing process".

In addition, the RedVerst has suggested a specific SWDP called *"co-verification"* process. The process unites the target software design with the formal specifications and the test suites development in a concurrent fashion. One of the valuable advantages of the process is the production of the test suites before the target implementation is completed. Another important benefit of the process is a clear scheme of cooperative work of architects, designers and test designers. The "co-verification" advantages provide good opportunities for early detection of design (the most costly) errors.

Lessons learned. From the one hand, the KVEST has demonstrated feasibility of SBT use in industrial applications. From the other hand the KVEST has been successfully deployed as technology for only regression testing. The customer has not undertaken roles of specification and test designers yet. Usually the similar problems of technology deployment are explained by resistance of users and managers in accordance to formal techniques as a whole. It is true, but there exist important reasons of the "resistance". The reasons are common for KVEST and for many other SBT technologies. The most significant reasons are as follows:



SBT technologies are usually bi- or three-lingual (a user has to write in specification, test design, and programming languages) ⟸ **Multi-lingual tools**

SBT tool user does not have direct instructions how to write specification and test scenarios, how to re-use these artifacts. Therefore, there is significant re-work during specification and testing evolving software. ⟸ **No instruction**

Difficulty of understanding structure of generated code of test suite components, problems in fitting the test suites for specific SBT user's needs ⟸ **Intricate generated code**

Problems of integration specification and test design tools with standard software development environment ⟸ **Non-coordinated tools**

How to introduce 'extra' works like formal specification in traditional work flow? ⟸ **'Extra' words**

5 Next Step — Which Feature First?

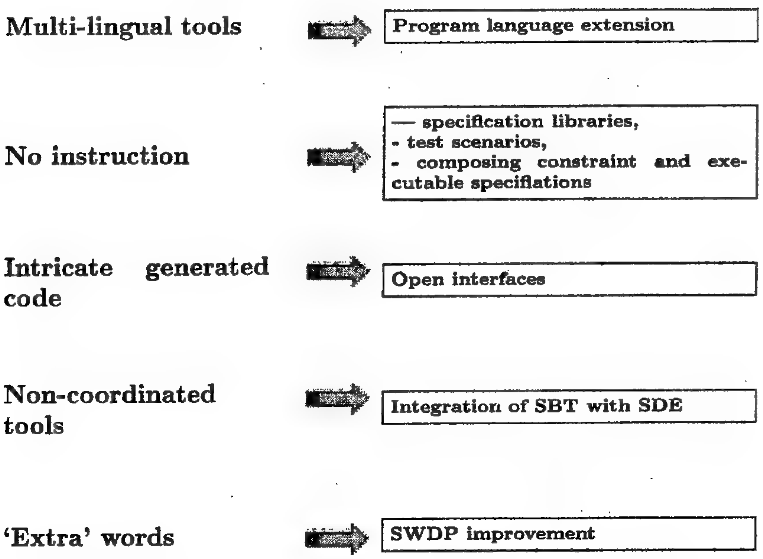
To overcome these five problems the five following solutions are suggested.

"Multi-lingual tools" problem It is first well-recognized problem in SBT: what specification language (notation) should be chosen in a practical project. There are two main alternatives in the notation choice: the formal specification languages (like classical ones) and **extensions of usual programming languages**. Both alternatives have advantages and drawbacks. The KVEST technology followed first way and has demonstrated feasibility of the approach. However now its becoming evident that second way promises more advantages in real-life industry context. The main argument for programming language extension vs. using formal specification language is the evident advantage of mono-lingual system against bi- or multi-lingual system. In addition, there are problems of involving software engineers in study and training in formal specification languages.

The idea of programming language extension for specification purpose is not novel one. The similar suggestions were discussed by B.Liskov, D.Parnas, and others in early 1970th. In the mid of 1990th C, C++, and Java were extended by joint X/Open Company Ltd. and the Sun Microsystems, MITI's Information-technology Promotion Agency group [22], Java by R.Krammer [10]. Eiffel [24],VDM-SL and VDM++ had originally facilities both for programming (prototyping) and for constraint specification.

Success of these extensions is quite restricted. Some of the tools/technologies are used only as in-house tools, others are mainly used in academic area. The reason of the obstacle is weakness and incompleteness of features provided to a practical specification and test designer. As an example, in more details the drawbacks of ADL and iContract are described in other paper presented in the proceedings and on RedVerst web-site [4, 27]. The RedVerst has designed UniTesK concept of SBT using programming language extension. The concept is presented in [15].

"No instruction" problem. The tutorial, monographs, and manuals are necessary but not sufficient materials for SBT propagation. In addition to these materials the software engineers need examples, prototypes, **libraries of specifications**. The OO approach opens new opportunity for architecture of these libraries [4, 15]. Because specifications are usually more abstract than implementation, so re-use of the specifications could be more simple. This opportunity is noted by Eiffel society [24], but they use one for only rapid prototyping. An additional, valuable advantage caused by introduction of formal specification in a software development process is the **test suite component re-use** because these components are generated from **re-usable specifications**. The RedVerst has developed the techniques for test suite re-use. First one is intended for C like software and uses template-based technology. The UniTesK approach expands the area of re-usable components and provides the OO techniques for representation of storing artifacts and integration of the handmade and the generated artifacts into the ready for use OO test suites.



Multi-lingual tools → Program language extension

No instruction → — specification libraries, - test scenarios, - composing constraint and executable specifiations

Intricate generated code → Open interfaces

Non-coordinated tools → Integration of SBT with SDE

'Extra' words → SWDP improvement

As mentioned above **executable specifications**, including FSM like specifications, are quite suitable for test sequence generation but have restricted possibilities for test oracle generation. There is an evident idea: to unite executable and constraint specifications to gain advantages of both these approaches. However, two obstacles prohibit from the union. First, it is doubling effort of specification design, and, second, there is a certain risk in developing the inconsistent parts of specifications. Some researchers try to derive executable specification from constraint specification automatically [12, 13]. It seems the idea is quite attractive but cannot be applied to any kind of real-life software. RedVerst has developed the techniques for replenishment of constraint specification with implicit FSM specifications. The idea of implicit FSM specification is briefly described in [6]. In detail the theoretical background of the technique is described in [1, 5]. New kinds of FSMs differ in degree of determinism and timing characteristics of reaction appearing. The variety of FSMs allows to generate the test sequences for wide spectrum of software including distributed and real-time applications. FSM based techniques allow to generate an exhaustive test (in sense of the model). Sometime (for example, for debugging) it is desirable to use non-exhaustive but some specific tests usually based on use cases or test scenarios. A union of scenario approach and FSM-based approach is used in UniTesK [4]. The technique allows describing a main idea (scenario) of a test case family and to generate several test cases (using the implicit FSM technique) that belong to this family.

"Intricate generated code" problem. It is a common problem of the tools generated a code from some sources; the intricate generated code is too complicated for understanding and debugging. A prospective solution is design an **open OO test suite architecture** where the generated and handmade artifacts are stored separately, in different classes, but are closely and naturally linked by means of usual relations used in OO design and programming. UniTesK presents an example of such OO test suite architecture [4, 15].

"Non-coordinated tools" problem. The UniTesK dream is integration with arbitrary SDE based on some standard interfaces. UniTesk requirements in the case are as follows. SDE should provide facilities for:

- SBT tools invocation from the SDE
- synchronization of the windows related to SBT input/output
- key words/syntax setting
- diagnostic messaging.

"'Extra' works" problem. Introduction of SBT implies appearance of new activities, roles, and artifacts in SWDP. The specifications could be presented as informal (for example, draft functional requirement specification), semi-formal (like UML use cases), and formal specifications. The new artifacts raise the necessity in new personnel, techniques, and tools — negative consequences. They allow well-organized and computer-aided requirements tracking, rapid prototyping, and test and documentation generation — positive consequences. So, to take an advantage of SBT an organization should invest some additional effort to compensate possible negative consequences of this prospective technology (it is true for any novel technology).

6 Conclusion

The paper presents an outline of current state of the art and prospective solutions of SBT problems. We focus on API specification testing because it is the base and most uniform level of software interfaces. This short review of SBT techniques did not consider whole variety of techniques known academic area. Our attention was only paid to the approaches have been used in real-life SWDP.

There is no any unified approach for specification and SBT and inside each of these approaches there is no any unique tool that performs all necessary actions. It is rather well. However, the known research results and commercial tools have shown feasibility of SBT approach in real-life application of arbitrary complexity. To be introduced in practice any technology must provide at least minimal set of features that meet the most needs, it is so-called critical mass. Above "Next step" solutions outline the critical mass. The era of toy examples and pioneer SBT projects is finishing.

References

1. Abstract Syntax Notation One (ASN.1) standard. Specification of Basic Notation. ITU-T Rec. X.680 (1997) — ISO/IEC 8824-1:1998.
2. S. Antoy, D. Hamlet. Automatically Checking an Implementation against Its Formal Specification. IEEE trans. On Soft.Eng, No. 1, Vol. 26, Jan. 2000, pp. 55–69.
3. D. Bjorner, C. B. Jones (editors). Formal Specification and Software Development. — Prentice-Hall Int., 1982.
4. I. B. Burdonov, A. V. Demakov, A. A. Jarov, A. S. Kossatchev, V. V. Kuliamin, A. K. Petrenko, S. V. Zelenov. — J@va : extension of Java for real-life specification and testing. — In these transactions.
5. I. B. Bourdonov, A. S. Kossatchev, V. V. Kuliamin, A. V. Maximov. Testing Programs Modeled by Nondeterministic Finite State Machine. — (see [27] white papers).
6. I. B. Bourdonov, A. S. Kossatchev, V. V. Kuliamin. Using Finite State Machines in Program Testing. "Programmirovanije", 2000, No. 2 (in Russian). Programming and Computer Software, Vol. 26, No. 2, 2000, pp. 61–73 (English version).
7. I. Bourdonov, A. Kossatchev, A. Petrenko, and D. Galter. KVEST: Automated Generation of Test Suites from Formal Specifications. Proceedings of World Congress of Formal Methods, Toulouse, France, LNCS, No. 1708, 1999, pp. 608–621.
8. I. Burdonov, A. Kossatchev, A. Petrenko, S. Cheng, H. Wong. Formal Specification and Verification of SOS Kernel. BNR/NORTEL Design Forum, June 1996.
9. R.-K. Doong, P. Frankl. Case Studies on testing Object-Oriented Programs, Proc. Symp. Testing, Analysis, and Verification (TAV4), 1991, pp. 165–177.
10. R. Kramer. iContract — The Java Design by Contract Tool. Fourth conference on OO technology and systems (COOTS), 1998.
11. S.-K. Kim and D. Carrington. A Formal Mapping between UML Models and Object-Z Specifications (http://svrc.it.uq.edu.au/Bibliography/bib-entry.html?index=851).
12. L. Murray, D. Carrington, I. MacColl, J. McDonald, P. Strooper. Formal Derivation of Finite State Machines for Class Testing. In Lecture Notes in Computer Science, 1493, pp. 42–59
13. K. Lerner, P. Strooper. Refinement and State Machine Abstraction. — SVRC, School of IT, The University of Queensland, Technical report No. 00-01. Feb. 2000 (http://svrc.it.uq.edu.au/).
14. Message Sequence Charts. ITU recommendation Z.120.
15. A. K. Petrenko, I. B. Bourdonov, A. S. Kossatchev, V. V. Kuliamin. Experiences in using testing tools and technology in real-life applications. — Proceedings of SETT01, India, Pune, 2001.
16. A. Petrenko, A. Vorobiev. Industrial Experience in Using Formal Methods for Software Development in Nortel Networks. — Proc. Of the TCS2000 Conf., Washington., DC, June, 2000.
17. D. K. Peters, D. L. Parnas. Using Test Oracles Generated from Program Documentation. IEEE Trans. on Software Engineering, Vol. 24, No. 3, March 1998, pp. 161–173.
18. N. Plat, P. G. Larsen. An Overview of the ISO/VDM-SL Standard. SIGPLAN Notes, Vol. 27, No. 8, August 1992.
19. J. Ryser, M. Glinz. SCENT: A Method Employing Scenarios to Systematically Derive Test Cases for System Test. — ftp://ftp.ifi.unizh.ch/pub/techreports/TR-2000/ifi-2000.03.pdf
20. Specification and Design Language. ITU recommendation Z.100.
21. The RAISE Language Group. The RAISE Specification Language. Prentice Hall Europe, 1992.
22. http://adl.opengroup.org/
23. http://www.csr.ncl.ac.uk/vdm/
24. http://www.eiffel.com
25. http://www.fmeurope.org/databases/fmadb088.html
26. http://www.ifad.dk
27. http://www.ispras.ru/ RedVerst/
28. http://www.omg.org/uml/

Java Specification Extension for Automated Test Development

Igor B. Bourdonov, Alexey V. Demakov, Andrey A. Jarov, Alexander S. Kossatchev,
Victor V. Kuliamin, Alexander K. Petrenko, and Sergey V. Zelenov

Institute for System Programming of Russian Academy of Sciences (ISPRAS),
B. Communisticheskaya, 25, Moscow, Russia
e-mail: {igor,demakov,jandrew,kos,kuliamin,petrenko,zelenov}@ispras.ru
Web: http://www.ispras.ru/~RedVerst/

Abstract. The article presents the advantages of J@va, a specification extension of the Java language, intended for use in automated test development. The approach presented includes constraints specification, automatic oracle generation, usage of FSM (Finite State Machine) model and algebraic specifications for test sequence generation, and specification abstraction management. This work stems from the ISPRAS results of academic research and industrial application of formal techniques [1].

1 Introduction

The last decade has shown that the industrial use of formal methods became an important new trend in software development. Testing techniques based on formal specifications occupy a significant position among the most useful applications of formal methods. However, several projects carried out by the RedVerst group [3, 12] on the base of the RAISE Specification Language (RSL) [6] showed that the use of specification languages like VDM or Z, which are unusual for common software engineer, is a serious obstacle for the wide application of such techniques in industrial software production. First, the specification language and programming language often use different semantics and may even use different paradigms, so a special mapping technique must be used for each target language. Second, only developers having special skills and education can efficiently use a specification language. The possible solution of this problem is the use of specification extensions of widely used programming languages.

This article presents J@va – a new specification extension of Java language. Several specification extensions of programming languages already exist. ADL [5, 7] and iContract [8, 9] are the most known of them. A few extensions have been used in industrial projects. Why we invent a new one?

Our experience obtained in several telecommunication software verification projects shows that the formal testing method used in industry should not only allow automated test generation but also possess features such as clear modularization, suitable abstraction level management, separate specification and test design, and the support of test coverage estimation based on several criteria [14]. These subjects did not receive sufficient attention in ADL and iContract languages and technologies. The absence of integrated solution explains the restricted use of these languages and related tools.

The problem of test automation can be split into the *oracle* generation and the test sequence generation subproblems. Because of the paper size restrictions the issues of automated oracle generation are outside the scope of this paper, therefore we refer the interested reader to the [3–5] works. Our approach (UniTesK technology) employs FSM models to solve the second problem of test automation (see [11] for details). Such a model can be viewed as a high abstraction of the specification of the unit under test, but in contrast with specification it is more dependent on implementation, environment and current testing objectives. We call the description of this model the *test scenario*. Test scenarios directly deal with test design while specifications describe the abstract functionality of target system.

2 Key Features of J@va Approach

In this section we present the J@va key features, explain their necessity or advantages, and compare them with ADL and iContract. We focus mostly on features not supported or supported insufficiently by J@va contenders. In particular, we do not consider the general software contract specification approach used in all mentioned languages [8, 13], exception specification methods and parallel processing support.

Specification of object state. In J@va each specification class can have ***invariants*** representing the consistency constraints on the state of an object of this class. In ADL and iContract the same effect can be obtained only by including such constraints into pre- and postconditions of all class methods.

Axioms and algebraic specifications. J@va provides constructs to express arbitrary properties of the combined behavior of target methods in an algebraic specification fashion. The semantics of J@va axioms and algebraic specifications is an adaptation of the semantics of RSL ones [6]. Axioms and algebraic specifications serve as a source for test scenarios development – they are viewed as additional transitions in the testing model. During testing we call the corresponding oracle for each method call in an axiom and then check the global axiom constraint. Similar constructs can be found only in specification languages and are absent both in ADL and iContract.

Test coverage description. This is an essential feature for testing and software quality evaluation. Test coverage analysis also helps to optimize the test sequence dynamically by filtering the generated test cases, because usually there is no need to call target method with parameters from the same domain more than once. The coverage consisting of domains of different specification behavior, called the *functionality coverage,* can be derived automatically from the postcondition structure. J@va also has several special constructs for explicit test coverage description. The explicit coverage description and functionality coverage derivation allow providing fully automatic test coverage metrics construction and test coverage analysis. Neither ADL nor iContract has facilities for test coverage description and analysis.

Abstraction level management. The ability to describe system on different abstraction levels is very important both in forward and reverse engineering. The support of abstraction level changing allows developing really implementation-independent specifications, to follow top-down design or bottom-up reverse engineering strategy. In J@va, specifications and source code are fully separated. Their interaction is provided by a special *binding code.* This code performs synchronization of the model object state with the implementation object state and translates a call of model method into a sequence of implementation methods invocations. This approach allows using one specification with several source code components and vice versa, it also ensures the modularity of specifications and makes their reuse possible. No other of known Java specification extensions provides such a feature. Larch [10] provides the infrastructure the most similar to the J@va one but supports only two-level hierarchy.

Test scenarios. Test scenarios provide the test designer with a powerful tool for test development. The scenarios can be either completely user-written or generated on the base of once written templates and some parameters specified by test designer. In general, a J@va scenario defines its own FSM-like model of the target system, called *the testing model.* A scenario defines the state class for this model and the transitions, which must be described in terms of sequences of target method calls. The testing model should represent a FSM, which can be obtained from the FSM representing the target system by removing some states and transitions, combining a sequence of transitions into one transition and subsequent factorization. One can find details of this approach, some methods and algorithms of testing model construction in [11], where they are formulated in terms of FSM state graph properties.

In a more simple case, test scenario represents the sequence of tested operation calls that can lead to some verdict on their combined work. The test constructed from such a scenario executes the stated sequence and assigns the verdict; it also checks the results of each operation with the help of the operation's oracle.

Among existing Java extensions, only ADL provides some constructs for test case generation. However complex tests, e.g. for a class as a whole, have to be written entirely in the target programming language. An essential shortcoming of this approach is the lack of state-oriented testing support that forces the test designer to spend considerable efforts to ensure the necessary test coverage.

Open OO verification suite architecture. The verification suite consists of specification, test scenarios, binding code, and Java classes generated from the specifications and the test scenarios. The set of classes and relations between these classes and between verification classes and target Java classes are well defined. The architecture is described in UML and is easy to understand by any Java software engineer. The openness of the architecture does not mean necessity of the generated code customization for optimization or other purposes (there are other well-defined flexible facilities for fitting verification suite). However the openness significantly facilitates the understanding and the use of the technology as a whole. ADL and iContract users could read (and reverse engineer) generated code, however the structure of generated test harness is considered a private issue of ADL/iContract translator and could be changed at any time.

3 Conclusion

To become applicable in industrial software production, an automated test development technology must support a set of features that constitute something like a critical mass. The critical mass should be not too huge to be introduced in real-life software engineering and at the same time it should be sufficient for usual needs of software engineers. The J@va tries to achieve the object. More detail description of J@va and J@va based technology are presented on [1].

References

1. http://www.ispras.ru/~RedVerst/
2. http://www.fmeurope.org/databases/fmadb088.html
3. I. Bourdonov, A. Kossatchev, A. Petrenko, and D. Galter. KVEST: Automated Generation of Test Suites from Formal Specifications. *FM'99: Formal Methods. LNCS,* volume 1708, Springer-Verlag, 1999, pp. 608–621.
4. D. Peters, D. Parnas. Using Test Oracles Generated from Program Documentation. *IEEE Transactions on Software Engineering,* 24(3):161–173, 1998.
5. M. Obayashi, H. Kubota, S. P. McCarron, L. Mallet. The Assertion Based Testing Tool for OOP: ADL2, available via http://adl.xopen.org/exgr/icse/icse98.htm
6. The RAISE Language Group. The RAISE Specification Language. Prentice Hall Europe, 1992.
7. http://adl.xopen.org
8. R. Kramer. iContract – The Java Design by Contract Tool. // 4-th conference on OO technology and systems (COOTS), 1998.
9. http://www.reliable-systems.com/tools/iContract/iContract.htm
10. J. Guttag et al. The Larch Family of Specification Languages. // IEEE Software, Vol. 2, No. 5 (September 1985), pp. 24–36.
11. I. Bourdonov, A. Kossatchev, V. Kuliamin. Using FSM for Program Testing. *Programming and Computer Software,* Official English Translation of *Programmirovanie,* No. 2, 2000.
12. A. Petrenko, I. Bourdonov, A. Kossatchev, and V. Kuliamin. Experiences in using testing tools and technology in real-life applications. Proceedings of SETT'01, India, Pune, 2001.
13. B. Meyer. Object-Oriented Software Construction. Second Edition, Prentice Hall, Upper Saddle River, New Jersey, 1997.
14. A. K. Petrenko. Specification Based Testing: Towards Practice. In this transactions.

Specification-Based Testing of Firewalls*

Jan Jürjens[1] and Guido Wimmel[2]

[1] Computing Laboratory, University of Oxford
Wolfson Building, Parks Road, Oxford OX1 3QD, Great Britain
phone: +44 1865 284104, fax: +44 1865 273839, e-mail: jan@comlab.ox.ac.uk
[2] Department of Computer Science, Munich University of Technology
TU München, 80290 München, Germany
phone: +49 89 289 28362, fax: +49 89 289 25310, e-mail: wimmel@informatik.tu-muenchen.de

Abstract. Firewalls protect hosts in a corporate network from attacks. Together with the surrounding network infrastructure, they form a complex system, the security of which relies crucially on the correctness of the firewalls. We propose a method for specification-based testing of firewalls. It enables to formally model the firewalls and the surrounding network and to mechanically derive test-cases checking the firewalls for vulnerabilities. We use a general CASE-tool which makes our method flexible and easy to use.

1 Introduction

The increasing connection of businesses and other organisations to the Internet poses significant risks: Attackers from the Internet may exploit vulnerabilities in the internal hosts connected to the Internet to gain unauthorised access to the corporate network. Due to the complexity of computer systems, it is impossible to protect an internal host just by making sure that it has no vulnerabilities.

This motivates the use of firewalls [CB94] to protect a network from the Internet, or subnetworks from each other. Firewalls are complex systems composed of several hard- and software components the correct design of which is difficult, in particular for networks that use more than one firewall (e. g. larger companies). However, testing firewalls is usually confined to applying simple check lists (e. g. [ES99]) possibly using specialised tools (such as [Fre98]).

We propose an alternative approach: we formally model a firewall system, and derive test sequences automatically from the formal specification — following the approach to specification-based testing of [Wim00, WLPS00, LP00]. Testing the firewall with these test sequences provides more confidence that the firewall implementation actually provides the desired protection, than ad-hoc testing, especially since the test-sequences are derived with respect to the actual network topology. Our approach is embedded in an easy-to-use CASE framework [HMR$^+$98]. Because of its generality, there are few restrictions on the model: Firewall rules need not be of a special form, *stateful* firewalls can be modelled etc. The network model is also flexible, allowing to model possible faults or Trojan horses (malicious code injected by attackers) at the hosts. Various scenarios, such as stress test, spoofing (source address forging), and policy violations, can be tested. Additionally, one can check the firewall specification with a model-checker.

2 Testing Firewalls

In our approach, we give a (possibly partial) description of a network behaviour that presents a potential threat. From this, a test-sequence is derived automatically which indicates how the system should react to this threat according to the specification. This test-sequence can then be used for actual testing of the firewall.

2.1 Example Network and Formal Model

We consider an example (in the following called the Network) similar to the one given in [BMNW99]. Each interface has its own IP-address. The DMZ (*demilitarised zone*) contains a web-server, a DNS-server and a mail-server.

Here we consider IP-based packet-filtering firewalls. The firewalls should implement *rule-bases.* Each rule specifies a *source*, a *destination*, a *service-group*, the direction of the packet, and the *action* to be taken (*pass*

* This work was partially supported by the Studienstiftung des deutschen Volkes, and by the German Ministry of Economics within the FairPay project

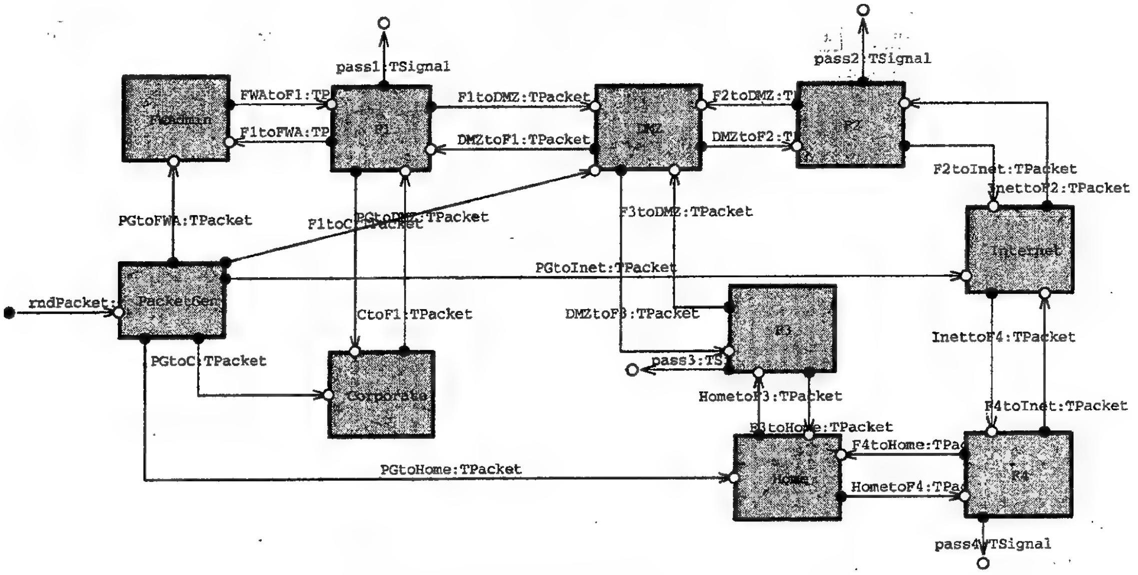


Fig. 1. SSD for firewall system

or *drop*). We specified a security policy for the example network and translated it into rules. For example, packets should pass the firewalls only if the direction of the packet complies with the topology of the network with respect to its source and destination.

We modelled the network containing the firewalls with help of the CASE tool AutoFocus/Quest [HMR+98, HMS+98], a tool for graphically specifying distributed systems. AutoFocus supports different views on the system model, describing structure, data types, behaviour and interactions. These views are related to UML-RT diagrams. In addition to modelling, AutoFocus offers simulation, code generation, test sequence generation and formal verification of the modelled systems.

The **structural view** on our example network system is depicted in Figure 1, as an AutoFocus system structure diagram (SSD). Each network component (subnets and firewalls) is an AutoFocus system component, drawn as a rectangle. These components can exchange data via named channels, which connect output and input ports (filled and empty circles). The packets sent through the network are modelled using hierarchical data types — in particular, the TPacket data type carries information about source and destination address and about the service.

Finally, each component in the network model is assigned a specified behaviour, using **state transition diagrams** (STDs). STDs correspond to extended finite state machines (meaning they can have a data state as well as a control state, and communicate with the other state machines). In our example, the STDs model the routing of messages and implement the firewall rules.

The state transition diagrams for the packet generator (which creates random packets to travel through the network) and the subnets are straightforward — in the subnets of our current network model, the packets are just relayed at random to other output ports. However, the model is very flexible in this respect, so the functionality could also be changed such that the packets may be manipulated. The STDs assigned to the firewall components represent the firewall rules.

2.2 Testing

In our approach for firewall testing, we use the formal AutoFocus model as a specification to generate test cases. We call this *specification-based test sequence generation* (see e.g. [WLPS00]). For this purpose, first test case specifications based on the system model have to be formulated. Test case specifications would be, for example, that we look for executions where a packet arrives at a certain interface of a component, or executions where a packet is dropped by a firewall. These are translated into logic and solved. The solutions are all test cases of a given maximum length satisfying the test case specification. These test cases represent concrete system executions (which exact packets with which data originate at which component, the way they travel etc.), can be depicted as message sequence charts and fed into the actual implementation of the firewall system for testing. The test sequences are computed based on constraint logic programming (CLP), see [LP00] for details.

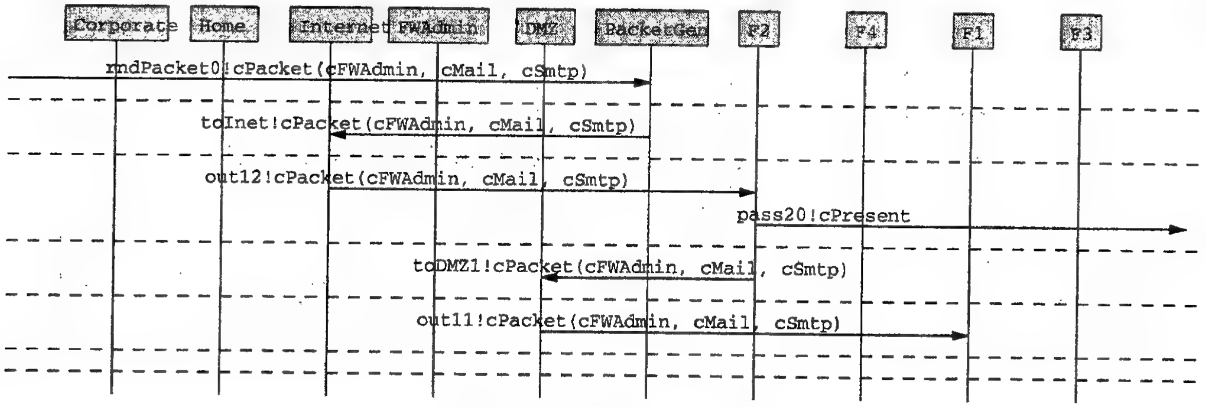

Corporate
Home
Internet
FWAdmin
DMZ
PacketGen
F2
F4
F1
F3
rndPacket0!cPacket(cFWAdmin, cMail, cSmtp)
toInet!cPacket(cFWAdmin, cMail, cSmtp)
out12!cPacket(cFWAdmin, cMail, cSmtp)
pass20!cPresent
toDMZ1!cPacket(cFWAdmin, cMail, cSmtp)
out11!cPacket(cFWAdmin, cMail, cSmtp)

Fig. 2. Test Case for firewall FW1

Thus, with our approach we can systematically generate many (or even all) test cases of a given maximum length to verify chosen security aspects of the firewall implementation. This leads to an improved reliability of the system resulting from the test, as opposed to ad-hoc testing.

In general, our approach supports all kinds of test scenarios that can be specified based on the execution history of the system. Important test scenarios for threats against the firewall example system we tested include the following:

- **Stress test.** For a chosen firewall component, generate all test cases from the specification, where this firewall should drop an arriving packet. Figure 2 shows a test sequence with route of a packet dropped by F1 as it violates the security policy.
- **Spoofing.** In this scenario, packets with forged source addresses are exposed to the system and it is tested if the firewalls behave correctly.
- **Policy violations.** As explained in [Gut01], firewall systems have to be based on a security policy. In [Gut01], this policy is given in the form that, if a packet *was in* a certain subnet, and *reaches* another subnet, a certain condition on its source and destination address and service has to be fulfilled. Test sequences can be generated that fulfill or violate these policies.

Systematic Selection of Test Sequences. For a given test case specification, the number of test cases can get fairly large — especially with more complex data types than in our example. We can use domain-specific knowledge to improve coverage, e.g. the maximum length of the test sequences can be restricted to the diameter of the network, or use conditions on the source or destination addresses.

3 Conclusion and Future Work

We proposed a method for specification-based testing of firewalls, enabling one to formally model the firewall and the surrounding network and to mechanically derive test-cases checking the firewall for vulnerabilities. We used a general CASE-tool which makes our method flexible and easy to use. We demonstrated our approach with an example firewall system.

In future work we will consider advanced network and firewall designs, such as authentication headers (using cryptography), virtual private networks, and distributed firewalls. It would be desirable to have a higher-level language for the security policies that is automatically translated into rules (following [Gut01]).

Also, our approach opens up the possibility to go beyond test-sequence generation and perform the testing automatically, on an actual firewall system.

References

[BMNW99] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: A novel firewall management toolkit. In *Security and Privacy*, 1999.

[CB94] W. Cheswick and S. Bellovin. *Firewalls and Internet Security: repelling the wily hacker.* Addison-Wesley, 1994.

[ES99] UK IT Security Evaluation and Certification Scheme. UK ITSEC Certification Report No. P117 – CyberGuard Firewall for UnixWare, 1999.

[Fre98] M. Freiss. *Protecting Networks with SATAN.* O'Reilly, 1998.

[Gut01] J. Guttman. Security goals: Packet trajectories and strand spaces. In R. Gorrieri and R. Focardi, editors, *Foundations of Security Analysis and Design*, LNCS. Springer, 2001. Forthcoming.

[HMR+98] F. Huber, S. Molterer, A. Rausch, B. Schätz, M. Sihling, and O. Slotosch. Tool supported Specification and Simulation of Distributed Systems. In *International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 155–164, 1998.

[HMS+98] F. Huber, S. Molterer, B. Schätz, O. Slotosch, and A. Vilbig. Traffic Lights – An AutoFocus Case Study. In *1998 International Conference on Application of Concurrency to System Design*, pages 282–294. IEEE Computer Society, 1998.

[LP00] H. Lötzbeyer and A. Pretschner. Testing concurrent reactive systems with constraint logic programming. In *2nd Workshop on Rule-Based Constraint Reasoning and Programming*, Singapore, 2000.

[Wim00] G. Wimmel. Specification Based Determination of Test Sequences in Embedded Systems. Master's thesis, Technische Universität München, 2000.

[WLPS00] G. Wimmel, H. Lötzbeyer, A. Pretschner, and O. Slotosch. Specification Based Test Sequence Generation with Propositional Logic. *Journal on Software Testing Verification and Reliability*, 10, 2000.

Software Construction

At the Edge of Design by Contract

Short Abstract

Bertrand Meyer

Interactive Software Engineering (Santa Barbara, USA)
Monash University (Melbourne, Australia)
e-mail: Bertrand_Meyer@eiffel.com

Combining the Hoare-Dijkstra concepts of assertions and systematic software construction with principles of object technology and data abstraction, Design by Contract provides a solid basis for building, testing, documenting and maintaining quality O-O software. This presentation will examine issues at the forefront of Design by Contract, including a number of problems to which no answer is known at the moment, covering in particular the areas of program correctness, component validation, and concurrency.

Academic vs. Industrial Software Engineering: Closing the Gap

Andrey N. Terekhov[1] and Len Erlikh[2]

[1] St.Petersburg State University, LANIT-TERCOM
Bibliotechnaya sq., 2, office 3386
198504, St.Petersburg, Russia
e-mail: ant@tercom.ru
[2] Relativity Technologies, Inc.
1001 Winstead Drive
27513, Cary, NC, USA
e-mail: lerlikh@relativity.com

Abstract. We argue that there is a gap between software engineering cultivated in the universities and industrial software development. We believe that it is possible to get academia and industry closer by starting projects that will require solution of non-trivial scientific tasks from one side and long-term commitment to create a product out of this research solutions from the other side. We illustrate our position on a real-world example of collaboration between an American company Relativity Technologies and research teams from St.Petersburg and Novosibirk State Universities. We also point out that the current economic situation in Russia presents unique opportunity for international projects.

Introduction

Industrial programming is usually associated with big teams of programmers, strict timelines and established solutions and technologies. On the other hand, the main goal of academic research is to find new solutions and break existing stereotypes. Unfortunately, amazingly low percentage of scientific results makes their way into practice, and even when they do, the process is very slow.

In the meantime, practice always required a solution of the tasks that are infeasible from the point of view of the existing theory. Today this common truth takes on special significance for software engineering because the number of its applications really exploded. It is a well-known situation when practitioner is posing a problem and theoretician is reasoning why this task is unlikely to be solved. But the proof of impossibility of correct solution of the problem does not satisfy the demand for it, so practitioners start to seek partial or heuristic solutions or try to use "brute force" method.

In this article we try to show that even on this shaky ground it is better to use specialists that know the theoretical restrictions, complexity estimations, optimization methods and other traditionally scientific knowledge. This sounds pretty obvious, but somehow the chasm between academic and scientific communities is very difficult to close. What are the main reasons for this?

It is well-known that software engineering is differing from pure mathematics or even computer science. Proved theorem or complexity estimation for some algorithm are results by themselves, and there are no other requirements for their creation other than scientists' talent, pen and paper. On the other hand, in software engineering a new interesting approach or even working prototype does not guarantee that they will lead to the successful and ready-to-use product. To achieve this, one should add up large teams, investments and strict industrial discipline.

We try to illustrate process of collaboration between industry and science on the example of creating an automated reengineering tool RescueWare, which automates reengineering of legacy software, i.e., conversion of systems written in COBOL, CICS, embedded SQL, BMS, PL/I, ADABAS Natural and other languages, working mostly on IBM mainframes to C++, VB, Java on Windows and UNIX platforms. Software reengineering does not end up in simple translation from one language to another — completlely different schemes of dialog with the user, access to legacy databases, recovery of lost knowledge about the program make this task much more difficult.

This project was carried out by large international team, which was geographically spread from North Carolina (USA) to St.Petersburg and Novosibirsk. The customer for this project was an American company Relativity Technologies, and the team that worked on this project included scientists from S.-Petersburg and Novosibirsk universities, LANIT-TERCOM company and Institue of Informatics Systems of Siberian division of Russian Academy of Sciences. The total investment in this system amounts to more than 400 man-years.

1 Architecture for Multi-Language Support

From the very beginning we were oriented on creation of *multi-language* translator, so one of our first tasks was to design a unified intermediate language (IL) for our system. The idea is that program transformation is two-staged: at first the program is converted to IL, and then to the target language. When there are M input and N output languages, this approach makes it possible to limit the amount of work to $M + N$ compilers instead of $M * N$ [1].

The problem apperas when notion of "natural program" in one language contradicts with the same notion in other language. For instance, using unconditional branch statements is usual for COBOL, but in Java there are no such statements at all. Some special transformations may be required to solve this problem. From semantics' point of view, we can name the following levels of program representation:

1. Control flow representation
2. Data flow representation
3. Representation of values

Thus IL must contain abstract means for program representation at all these levels, and the transformation will look as follows: first language constructs of the source language are "raised" to abstract intermediate representation and then they are "lowered" to concrete representation in the target language. The degree of abstraction should be carefully chosen to make sure that this lowering down leads to natural projections to the target language.

The weak point during IL design is the choice of data types and standard operations. While the set of control constructs in different languages is more or less suitable for unification, data types system could be significantly different. In the meantime, it is inexpedient to simply combine all types of the source languages, because addition of new language will require major changes to existing compiler functionality.

We believe that this task presents a good example of semantic gap between academic research and industrial programming. The idea of unified IL makes sense only in large-scale projects, and these projects are out of academic scope. On the other hand, average programmer in the industry just does not possess all the knowledge, which is required for successful implementation of this approach.

Note that "naturality" of IL structure, which was mentioned above, is also a good example of difficult to formalize notions. These notions are often necessary to solve usual everyday tasks. Another example of difficult to formalize notions is the definition of "good program" criteria [2].

2 Re-Modularization. Class Builder

Another interesting task that we encountered during creation of automated reengineering tool is remodularization of programs into components, modules or classes [3,4].

This task could be described as follows: there is a large application that consists of multiple files, which contain declaration of data and procedures. Variables and procedures from different files are interacting with each other through some external objects, which we called *dataports*. For legacy systems usual dataports are CICS statements, embedded SQL and other infrastructure elements.

This task was formalized as follows: application was represented as a graph with application objects as junctions of various types (variable, procedure or external object), and relations between them as graph edges. Edges are also typed (for instance, procedure call, variable usage in procedure, working with external object through variable etc.). Also, all edges are attributed with some numbers, which defines the "power" of this relation. For example, the power for procedure call relation could be defined by the number of parameters passed: the more parameters we have, the more we want to place both callee and caller to one component.

This graph should be divided into some areas of strong connectivity. To do this, we introduce the notion of gravity between edges, which is calculated as sum of powers of all edges connecting them multiplied to coefficient defined by the edge type, minus some constant, which is defined by the pair of edge types.

Then by complete enumeration we find those junction sets, for which the sum of gravity force between themselves and the junctions from other sets are maximal (of course, gravity forces with the junctions of other groups are taken with minus sign).

It is clear that this good idea will not work in real life, because the number of graph junctions in real applications is way too much to use exhaustive searches. But we managed to find some heuristic approaches, which made it possible to achieve practical results.

First of all, we fixed some coefficients for different types of edges and repulsive forces for different types of junctions. However, the user can assign coefficients on his own if he believes this to be of importance for his application.

Secondly, in the complete graph of application we will start with sub-graph, which consists of the junctions corresponding to external object plus edges and junctions of any other types, which connect these external objects. The heuristics is that we believe external objects to be cross-linking and thus we add repulsive forces only for them. On the other hand, if two external objects are using a lot of common variables and procedures, then nothing prevents them from ending up in one component. We also used some other optimizations of this algorithm.

3 Program Slicing

Let us suppose that a legacy system performs ten functions, seven of which are no longer needed, but the remaining three are in active use, and as it often happens with legacy programs, nobody knows *how* these three functions work. In this case it is necessary to create a tool for deep analysis of the old programs, which can help maintenance engineer to find and pick out the required functionality, put the corresponding parts of the program into a separate module and reuse it in the future, for instance, to move it to modern language platform.

The solution of this task is based on creating static slices of the program and their modifications. We regard program slices to be a subset of program statements that presents the same execution context as the whole program. In other words, slice is a program that contains the given statement and some other statements of the initial program, namely those that are related to this statement.

The following methods are implemented in RescueWare for automation of business rule extraction (BRE):

- Computational-based BRE
- Domain-oriented BRE
- Structure-oriented BRE
- Global BRE

All these methods assume generation of syntactically correct independent programs that preserve the semantics of the original code fragments.

Computational-based BRE forms the functional slice of the program, based upon the execution path and data definitions that are required to calculate the values of the given variable in the certain point of the program (see detailed description of this approach in [5]).

Domain-oriented BRE generates functional slice of the program, which is received by fixing the value of one of the input variables. Being based on theory of program specialization, domain-oriented BRE is best suited to separate calculations with many transactions and mixed input, into a series of "narrowly specialized" business rules with only one transaction per rule.

Structure-oriented BRE makes it possible to divide the programs written as a single monolith into several independent business rules, based on the physical structure of the initial program. Also, an additional program is generated that calls the extracted slices in a proper sequence and using the correct parameters (ensuring the semantic equivalence of this program to the initial one). This method is best suited to divide old large programs into parts that are easier to handle.

Finally, global BRE helps to apply these methods to a number of programs simultaneously, and thus supports BRE on system-wide basis.

4 Conclusion

As of right now, products such as RescueWare are not really typical for the market, because creation of RescueWare required solution of *many* scientifically difficult problems. Let us briefly mention other achievements: relaxed parser that ensures collection of useful information even for quite distant dialects of the language, different variants of data flow analysis, using sophisticated algorithms of pattern matching for identification of structure fields in PL/I etc.

Finally, we would like to emphasize that Russia is in a special position to make this vision come true, because it has an undoubted advantage in the level of education at the software market. We hope that our experience of successful cooperation of American company with Russian scientists could serve as a good example for many Western companies.

References

1. A.P. Ershov "Design specifications for multi-language programming system", Cybernetics, 1975, No. 4. P. 11–27 (in Russian)
2. I.V. Pottosin "A "Good Program": An Attempt at an Exact Definition of the Term" // Programming and Computer Software, Vol. 23, No. 2, 1997. P. 59–69.
3. A.A. Terekhov "Automated extraction of classes from legacy systems" // In A.N. Terekhov, A.A. Terekhov (eds.) "Automated Software Reengineering", S.-Petersburg, 2000, P. 229–250 (in Russian)
4. S. Jarzabek, P. Knauber "Synergy between Component-based and Generative Approaches", In Proceedings of ESEC/FSE'99, Lecture Notes in Computer Science No. 1687, Springer Verlag, P. 429–445.
5. A.V. Drunin "Automated creation of program components on the basis of legacy programs" // In A.N. Terekhov, A.A. Terekhov (eds.) "Automated Software Reengineering", S.-Petersburg, 2000. P. 184–205 (in Russian)

A Method for Recovery and Maintenance of Software Architecture

Dmitry Koznov, Konstantin Romanovsky, and Alexei Nikitin

St.–Petersburg State University, Faculty of Mathmatics and Mechanics
Department of Software Engineering
198504 Bibliotechnaya sq., 2, St.-Petersburg, Russia
e-mail: {dim,kostet,lex}@tepkom.ru

Abstract. This paper proposes a method for recovery and subsequent maintenance of the architecture for actively evolving software systems. The method's underlying idea has to do with constructing a basic set of the architecture elements, which set would then be used for creating different views of the system. The responsibilities of the modules, which make up the software system, as well as elements of the system's data dictionary, are considered elements of this basic set. Of special meaning is the fact that the system is being actively maintained and developed, that the knowledge about it is accessible, but needs to be alienated from the respective bearing media, to be generalized and formalized.

1 Introduction

While many software products, for which the architecture was never formalized, may exist for years and be successfully maintained, still one can face a situation when formalization of the system architecture becomes a priority task. As a result, a lot of architectural imperfections in the system reveal themselves, and so there comes such moment in the life of the product, when its further development is impossible without improving the architecture, which requires formalizing the latter. Even various methods of analyzing and designing software systems have been spreading widely [11, 9, 10], it is a greate problem to use such methods in the situation like this. On the one hand this activity can cause serious internal restructuring of the system. On the other, we cannot ignore the commercial aspects of the software development process, issuing of new versions, service packs, and the implementation of new customer's requests.

Thus there is a problem: how to perform reverse engineering of the architecture of the actively evolving system, with most effective gathering and formalisation of the meta-information on the system. In the same time the architecture views should be supported in actual while the system will evolve.There are many methods and tools of reverse engeeniring designed for recovering the knowledge about a system architecture based on source code parsing (Refine/C, Imagix 4D, Rigi, Sniff [6, 9], RescueWare). Also there are formal methods of the reverse engineering [3].

All these methods have the same week point: absence of mechanism for synchronization of the model recovered with the source code of the software system. This problem makes very ineffective the use of such methods for actively evolving systems. From this point of view, there are more perspective Use-case driven methods [4], because the models that were derived with this method should be more stable to the system's changes on the practice. Data-mining methods [7], relying on the inner links between system elements are also interesting from the same viewpoint. However, all these methods are more suitable for the architecture of the "dead" system.

Round-Trip development methods, that are provided with some CASE-packages (e.g. Rational Rose, Together), enables the bi-directional connection of source codes and the architecture view. But the quality of the information visualized is unsatisfactory, because in fact, we do not get any new meta-information on the system, but only visualize the source code structure.

2 Starting Point

Let's formulate the basic problems that are to be solved with the presented method of architecture recovery:

1. The utilization of the features of the "living" system for the most effective architecture recovery;
2. The ability of the maintenance and further development of the architecture view of the software system;
3. The ability of step-by-step adopting of the process of development and support of the architecture without any serious damage to industrial requirements to the process.

3 The Method

3.1 Structure of the Model

The system architecture model proposed with this method consists of the following views: Structure views, Dynamic views and Physical view.

The basis of describing the system architecture is a System Structure description, which it is proposed to implement on the physical level (Physical view) and the logical level (Structure views). The Dynamic views are needed in order to determine the main scenarios of the system's operation.

The necessity of different kinds of views of the software is well known [1, 5, 8]. With the method in discussion, it is proposed to construct also a set of various Structure and Dynamic views, which is motivated by the following considerations:

1. The participants of the project, whose positions are on different levels of the hierarchy (managers of different levels) need information about the system to be specially adapted;
2. There exist both a vertical division of the system (by business functions) and a horizontal one (by tiers – e. g., User Interface, Business-Logic, Data Access);
3. There are different modes of packaging and deployment of the system;
4. In order to compile the entire project, information about the structure of storing the source codes of the system is needed;
5. Organizational structure of the enterprise affects decomposition of the system.

Structure Views. It is proposed to organize Structure views in the form of UML class diagrams. We herewith associate some part of the system (subsystem) with a class. The subsystems are organized into a multiple containment hierarchy, with the only restriction that the aggregated subsystems become invisible from the context, in which their aggregate exists. The different views should be constructed upon the same basic set of subsystems, but in other respects do not require any special matching. Associations between the subsystems, which reflect their semantic connections, are possible on each level.

Dynamic Views. With the help of the dynamic views it is proposed to represent the main scenarios of the system's operation. One can associate with each level of a structure view a dynamic view, which would explain how the subsystems interact with each other. For this, it is proposed to use the UML Collaboration Diagrams.

Physical View. This view is designed for inventory of the software source codes and for associating them with elements of the structure and dynamic views. In the view, the following items are considered:

1. Set of program modules (e. g., for Microsoft Visual C++ these are projects);
2. System data dictionary, which consists of:
   (a) Persistent data (logical data of the system and the corresponding physical media);
   (b) Channel data, with which subsystems exchange not through persistent-structures (physically media are absent, only logical elements of the data are there);
   (c) Configuration data that constitute a kind of persistent data, but are responsible for tuning the system algorithms (logical data and the corresponding physical. media).

3.2 Constructing The Basic Set

The foundation of this method consists in constructing a basic set of subsystems, from which different structure views will be built. In order to keep the correspondence of the views to each other and to make the maintenance easier, all elements of such views are to be subsets of the basic set of subsystems. So, the views themselves are hierarchical coverages of the basic set: the elements of each view form an aggregation hierarchy and the set of leaf nodes in that hierarchy is a usual coverage of the basic set. The basic set is constructed from responsibilities, which are assigned to the system modules (3–5 responsibilities per each module) and with elements of the system data dictionary.

Elements of the basic set of subsystems may be included in subsystems of different views; therefore, for them multiple containment is permissible.

When construction of the basic set is finished, all participants of the development process may build for themselves a package of structure and dynamic views that they need. A coordination of different views is provided by due to integrity of the set of basic elements, which are placed on different diagrams.

4 Conclusions and Futher Research

The method presented is designed for recovery, formalization and further maintenance of architecture of the actively evolving software systems.

At the moment there are some open problems to focus the investigation on. These problems are mainly about the adoption of the method presented in the industrial process of software development:

1. Clear mapping of the presented method's notions to UML;
2. Organization procedure of the adoption of the method presented.

References

1. J.Rumbaugh, I.Jacobson, G.Booch. The Unified Modeling Language Reference Manual. — Addison-Wesley, 1999.
2. Daniel Aebi: Data Re-Engineering — A Case Study. — ADBIS 1997, 305–310
3. Liu, H. Yang H. Zedan: Formal Methods for the Re-Engineering of Computing Systems: A Comparison // X. COMPSAC '97 – 21st International Computer Software and Applications Conference, – 1997
4. Christian Lindig, Gregor Snelting: Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis. — ICSE 1997, 349–359
5. Philippe Kruchten : 4+1 view model of architecture. – IEEE Software, November 1995, 42–50
6. Berndt Bellay and Harald Gall : A Comparison of four Reverse Engineering Tools // Working Conference on Reverse Engineering (WCRE '97) — October 6–8, 1997
7. K. Sartipi, K. Kontogiannis, F. Mavaddat : Architectural Design Recovery using Data Mining Techniques. // IEEE European Conference on Software Maintenance and Reengineering (CSMR 2000), pages 129-139, 29 Feb.–3 March 2000, — Zurich, Switzerland, 2000
8. Philippe Kruchten: The Rational Unified Process. — Addison Wesley Longman, Inc, 1999
9. M.N. Armstrong, C. Trudeau: Evaluating Architectural Extractors. — IEEE Software 1998, 30–39
10. Ivar Jackobson: Object-Oriented Software Engineering. — Addison-Wesley, 1993
11. Grady Booch: Object-Oriented Analysis and Design. — The Benjamin/Cummings Publishing Company Inc., 1994

An Empirical Study of Retargetable Compilers

Dmitry Boulychev and Dmitry Lomov

St.-Petersburg State University, Faculty of Mathematics and Mechanics
Department of System Programming
198504, Russia, St.Petersburg, Bibliotechnaya sq., 2
phone/fax: +7(812)428-71-09, e-mail: {db,dsl}@tepkom.ru

Abstract. The paper describes evaluation results of some modern retargetable codegeneration frameworks. The evaluation was performed to estimate applicability of these approaches in hardware-software codesign domain so ease of retargetability and efficiency of generated code were main criteria. Evaluated tools were selected from National Compiler Infrastructure (NCI) project.

1 Introduction

Hardware-software codesign is modern technique aimed to obtain high productivity of real-time and embedded systems. Key feature of this approach is simultaneous development of the program and the target processor or specialization of parameterized processor architecture to match target software application.

Generally, codesign implies iterative development. Each iteration consists of building new hardware description based on previous profiling and efficiency estimations, building (somehow) compiler, debugger, simulator, compiling and possible debugging target application, profiling and estimation of profit/loss. So building set of retargetable tools is basic and very frequent procedure.

Despite a number of retargetability techniques building of compiler still remains matter of art. Since main code-generation approaches are investigated well the contiguous tasks (supporting of calling and linking conventions, building debugger and profiler etc.) should be solved (semi)-manually. The most crucial problem of building machine-dependent code optimizer also remains open.

Here we describe most recent retargetable code-generation frameworks that look most preferable for purposes under considerations and present the results of their evaluation.

2 Retargetability Issues

Compiler's retargetability is usually understood as its ability to be re-targeted to another machine platform "automatically" or "nearly automatically". This implies building of codegenerator from some description. Ideally such a description should be extracted from description of actual hardware but as for now there is well-known semantic gap between hardware description and codegenerator description. So now transition from hardware to codegenerator is mainly proceeds as follows: first verbal instruction set description is produced, then codegenerator description is written from it.

Starting from the most fundamental results in code generation area [1, 2] main retargetability technique stays tree pattern matching and dynamic programming. A number of ways to exploit this idea are investigated [4, 5, 10, 13, 14]; also there are a number of compilers based on them. These methods often considered as means of *instruction selection* so register allocation and instruction scheduling should be done separately.

Similar attribute-grammar based method described in [11]. Most of heuristic codegenerators use this notion.

3 Criteria and Methods

The basic factors to be taken into account are, of course, quality of generated code and ease of retargetability.

To assess ease of retargetability, each tool evaluated has been ported to a "toy" instruction set, designed for a specific algorithm.

To assess quality of generated code, we compare the performance of several benchmarks on architectures that the tools being evaluated are already ported. We use Intel 386 and Sun SPARC processors for this purpose.

We used benchmarks developed by Standard Performance Evaluation Corporation (SPEC)[1]. This is an industry-standard set of benchmarks to assess quality of computer systems.

[1] http://www.spec.org

4 Evaluated Tools

We selected compilers from *National Compiler Infrastructure (NCI)*[2] project. The project was started under support of DARPA and NSF by major USA Universities (Harvard, Princeton, Stanford, Rice etc.)

On the other hand we have chosen legendary GCC compiler [16] as most authoritative industrial optimizing C compiler. Evaluation of GCC was performed merely for comparison purposes.

NCI project is aimed at developing interoperable framework for constructing retargetable, optimizing compilers. Combination of these two qualities – *retargetability* and *optimization* – is crucial for hardware-software codesign. Without good retargetability, co-design cycle becomes unbearably long; without optimization, the whole idea of co-design is compromised, as non-optimizing compiler does not employ features of the target architecture to its best. NCI project compilers represent current state-of-the-art in developing easily retargetable, optimizing compilers.

Currently three C compilers are available from NCI: SUIF(MachSUIF), lcc and VPO-based compiler. We evaluated all of them.

SUIF and MachSUIF. SUIF *(Stanford University Intermediate Format)* [12] and MachSUIF *(Machine SUIF)* [15] are developed in Stanford and Harvard Universities correspondingly. Both systems are parts of NCI project.

VPO-based compiler. VPO *(Very Portable Optimizer)* is a part of Zephyr[3] project. The project is in turn part of NCI.

lcc compiler. lcc compiler was developed in Princeton University, USA, since 1991 and later was also involved into NCI project [6–9].

5 Results and Conclusions

Neither SUIF nor VPO turned out to be ready-to-use compilers — during the evaluation we encontered lots of bugs that had to be fixed.

Our benchmarks show that SUIF/MachSUIF compiler is competely unapplicable for producing efficient code. This is largely due to inappropriate instruction selection techniques and lack of optimizations.

As the ease of retargeting, `lcc` turned out to be the best of all considered tools. `gcc` and `VPO` on the whole show same level of retargetability, although `gcc` is much better documented.

Regarding the efficiency of generated code, we saw that generally `gcc` with optimizations on beats all the other tools on both platforms. If optimizations are turned off in all tools, `lcc` shows best performance. `VPO` has shown quite irregular performance — on some benchmarks it produces the best code of all, while on others it lose even to non-optimizing `lcc` compiler.

Finally, for each tool (except SUIF) we discovered "its favorite benchmark" — the one that the tool generates best code of all for.

We conclude that none of the methods considered allows to build a retargetable code generator that can directly be utilized for co-design purposes.

We also see the importance of instruction selection — `lcc`, a non-optimizing compiler with good instruction selection algorithm based on BURS, shows quite good performance.

However, good instruction selection is not enough for obtaining optimized code. `VPO` outperforms `lcc` on majority of tests.

This research shows the directions for further development in co-design and code generation area. Easily retargetable, optimizing compilers are vital for hardware-software co-design, but we see that techniques for building them are yet to be created.

Acknowledgments

We would like to thank Mikhail Smirnov and Eugene Vigdorchik — our colleagues in OOPS team of System Programming Department in SPbSU — for the invaluable discussions and assistance in obtaining the results presented in this paper.

[2] http://www.cs.virginia.edu/nci/

[3] http://www.cs.virginia.edu/zephyr

References

1. Alfred V.Aho, S.C.Johnson. Optimal Code Generation for Expression Trees. Journal of the ACM, Vol. 23, No. 3, July 1976, 488–501
2. Alfred V.Aho, Steven W.K.Tjiang. Code Generation Using Tree Matching and Dynamic Programming. ACM Transactions on Programming Languages and Systems, Vol. 11, No. 4, Oct. 1989, 491–516
3. Hubert Comon, Max Dauchet et al. Tree Automata Techniques and Applications. http://l3ux02.univ-lille3.fr/~tommasi/TATAHTML/main.html
4. H. Emmelmann, F.W.Schröer, R.Landwehr. BEG — a Generator for Efficient Back Ends. Proceedings of the SIGPLAN'89 Conference on Programming Languages Design and Implementation, 1989, 227–237
5. M. Anton Ertl. Optimal Code Selection in DAGs. Proceedings of the 26th ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages, 1999, 242–249
6. Christopher W.Fraser, David R.Hanson. A Retargetable C Compiler: Design and Implementation. Addison-Wesley Pub Co., Jan. 1995
7. Christopher W.Fraser, David R.Hanson. A Retargetable Compiler for ANSI C. ACM SIGPLAN Notices, Vol. 26, No. 10, Oct. 1991, 29–43
8. Christopher W.Fraser, David R.Hanson. A Code Generation Interface for ANSI C. Software — Practice and Experience Vol. 21, No. 9, Sept. 1991, 963–988
9. Christopher W.Fraser, David R.Hanson. Simple regigster spilling in retargetable compiler. Software — Practice and Experience, Vol. 22, No. 1, Jan. 1992, 85–99
10. Christopher W.Fraser, David R.Hanson, Todd A.Proebsting. Engeneering a simple, efficient code generator generator. ACM Letters on Programming Languages and Systems, Vol. 1, No. 3, Sep. 1992, 213–226
11. Mahadevan Ganapathi, Charles N. Fischer. Affix grammar driven code generation. ACM Transactions on Programming Languages and Systems, Vol. 7, No. 4, Oct. 1985, 560–599
12. Monika Lam et al., An Overview of the SUIF2 Compiler Infrastructure. Computer Systems Laboratory, Stanford University, 2000, http://suif.stanford.edu/suif/suif2
13. Eduardo Pelegrí-Llopart, Susan L.Graham. Optimal Code Generation for Expression Trees: An Application of BURS Theory. Proceedings of the conference on Principles of programming languages, 1988, 294–308
14. Todd A.Proebsting. BURS Automata Generation. ACM Transactions on Programming Languages and Systems, Vol. 17, No. 3, May 1995, 461–486
15. Michael D. Smith, Glenn Holloway. An Introduction to Machine SUIF and Its Portable Libraries and Optimizations. Division of Engineering and Applied Sciences, Harvard University, 2000, http://www.eecs.harvard.edu/hube/research/machsuif.html
16. Using and Porting GNU Compiler Collection (GCC). http://gcc.gnu.org/onlinedocs/gcc_toc.html

Data & Knowledge Bases

Conceptual Data Modeling: An Algebraic Viewpoint

Kazem Lellahi

LIPN, UPRES-A 7030 C.N.R.S
Université Paris 13, Institut Galilée,
93430 Villetaneuse France
fax +33 (0)1 4826 0712, e-mail: `kl@lipn.univ-paris13.fr`

Abstract. Conceptual modeling of a system consists of giving a structured form of information in a way it captures, as much as possible, the semantics of real word objects. Despite some formalization attempts, most conceptual techniques remain rather informal. Our aim in this paper is to provide an algebraic framework for the basic concepts of Entity-Relationship model. We believe that our approach can help designers in schema validation.

Keywords: Algebraic Semantics, Algebraic Specification, Conceptual model, meta-modeling.

1 Introduction

The most widely-used conceptual model for designing an operational database is the Entity Relationship model, or ER-model, originally introduced by Chen [3]. Conceptual modeling within this model concerns designing an application by means of *entity types* and *relationship types*. An entity type represents a set of real-world objects and a relationship type relates two or several entity types. A property of an entity (or a relationship) type is called an *attribute*. Each attribute, entity type, or relationship type is recognized by a unique identifier called its *name*. In order to capture more semantics of real-world objects and their relationships, those names are enriched with some constraint declarations. The most common constraints are *key constraint* and *cardinality constraint*. A key constraint concerns one entity type and a cardinality constraint one relationship type. In ER-literature, other kind of constraints have been proposed which concern several relationships types. Attributes, entity types, relationship types and constraints provide a specification of the application, representing its *conceptual level*. Several approaches have been proposed for representing this specification graphically. Such a graphical representation is called an ER-*diagrams*. One of the most important steps of designing an operational database is to determine its ER-*diagram*. A CASE (Computer Aided Software Design) tool helps to draw an ER-diagram and generate automatically an operational database schema[1]. Real world objects of an application at a moment of time may be viewed as the *physical level* of the application. It forms a *valid instance* of the ER-diagram. In the majority of proposed approaches, the border between conceptual and physical levels is blurred. This paper proposes an algebraic framework tracing this border neatly. To do this, we define formally, by operations and axioms, the conceptual level called ER-*schema*. Axioms are algebraic counter parts of constraints, and a valid instance is a finite model of these operations and axioms. To define formally the physical level, we interpret each attribute by a set of values called its *domain* and then we naturally extend this interpretation to entity types and relationship types. This separation of conceptual and physical level aligns our approach with classical algebraic specification of data types. Following traditional algebraic approaches, the class of all valid instances of a schema is called its "loose semantics". Moreover, we prove that the design process itself can be structured

[1] For example, POWER PC (a Sybase product) transforms an ER-diagram into a relational schema coded in SQL.

as a schema – the *meta-schema* – such that the loose semantics of the meta-schema is the class of all well-formed schemas. Note that the domain of an attribute in this approach can be a set of any kind of values, i.e., atomic values, collection values or tuples (records).

The remainder of the paper is organized as follows. Section 2 briefly describes the basic mathematical background and develops (max,min)-cardinality in a general and formal way. Section 3 introduces our formal definition of an ER-schema and compares it with classical approaches. We show how a schema can be seen as an algebra of a specification. Section 4 introduces an instance of such a schema in the style of denotational semantics. Section 5 explores a particular schema called the meta-schema, and proves that each valid instance of the meta-schema is an ER-schema in our sense. Section 6 briefly compares our results with some related works and draws a brief conclusion and indications to further research.

2 Constraints

For a partial function $f : X \to Y$ we call X its *source* and Y its *target*, and we denote by $def(f)$ its definition domain. The function f is called:

- *finite* iff X and Y are finite sets,
- *surjective* iff $f(def(f)) = Y$, and *total* iff $def(f) = X$

For two finite functions with the same source, $f : X \to Y$ and $g : X \to Z$, we say:

- f and g are *exclusive*, denoted $f \cap g = \emptyset$, iff $def(f) \cap def(g) = \emptyset$,
- f and g *cover* X , denoted $f \cup g = X$, iff $def(f) \cup def(g) = X$,
- f is *less defined than* g, denoted $f \sqsubseteq g$, iff $f(x) = g(x)$ for all $x \in def(f)$.

It is self evident that the problem of checking each of above constraints for finite partial functions is a decidable problem.

Let *nat* be the set of nonnegative integers and $\omega = nat \cup \{N\}$, where N is a symbol not in *nat*. We extend the usual order of *nat* to ω by $n < N$ for all $n \in nat$. Ordered pairs $(0, k)$ and (k, N) are said to be *basic cardinalities* if $k \in nat$. For instance, the classical (min, max)-cardinalities $(0, 1)$, $(0, N)$ and $(1, N)$ are basic. Now, we specify two binary operations $\wedge$ and $\vee$ and we define the set CARDINALITY of all cardinalities as follows:

– every basic cardinality is a cardinality, and

– if c_1 and c_2 are cardinalities, then so are $c_1 \wedge c_2$ and $c_1 \vee c_2$.

The semantics of each cardinality c, denoted $[\![c]\!]$, is defined as follows:

$[\![(0,0)]\!] = \emptyset$; $[\![(0,k)]\!] = \{x \mid x \leq k\}$, where $k > 0$

$[\![(k,N)]\!] = \{x \mid k \leq x \text{ and } x \neq N\}$, where $k > 0$.

$[\![c_1 \wedge c_2]\!] = [\![c_1]\!] \cap [\![c_2]\!]$, $[\![c_1 \vee c_2]\!] = [\![c_1]\!] \cup [\![c_2]\!]$

As a result, *relaxed* cardinality and *int-cardinality* constraints (following [11], [7]), are cardinalities in our sense. In particular, the classical cardinality $(1, 1)$ is the cardinality $(0, 1) \wedge (1, N)$, and for any a and b in *nat* such that $0 \leq a \leq b < N$, the interval cardinality (a, b) is the expression $(a, N) \wedge (0, b)$. But our cardinality expressions also define other kinds of cardinalities. For example, the expression $(0, k) \vee (k_1, N)$ with $k < k_1$ is a new kind of cardinality that we call "at most k or at least k_1". It corresponds semantically to the set $\{n \mid 0 \leq n \leq k\} \cup \{n \mid k_1 \leq n\}$.

Roughly speaking, CARDINALITY can be viewed as a data type specified by two operations $\wedge$ and $\vee$. Then $[\![\,]\!]$ defines an algebra of that specification.

3 Conceptual Schema

Conceptual modeling of data obeys some rules called *meta-rules*. Meta-rules concern generic concepts and are independent of any applications. For instance, generic concepts of entity-relationship model include entity, attribute, relationship, domain, cardinality and label (role). Meta-rules for this model may include the following:

Rule 1: Every entity type has at least one attribute. Some attributes of an entity type may be declared as key attributes.

Rule 2: Every relationship type is at least binary. Each entity type participating in a relationship is accompanied with a (min, max)-cardinality and a role. If an entity type participates in a relationship type twice, the corresponding roles must be different.

Rule 3: Overloading attribute names is not allowed, and every attribute name is either an entity type attribute name or a relationship type attribute name, but not both.

The core of a conceptual data modeling tool consists of an implementation of such rules. Any formalization of such rules can help designers of such systems. In what follows, we shall give a rigorous formalization of these

rules in terms of simple mathematical concepts given in the previous section.
Let ATT, ENT, REL, and LAB be enumerable pairwise disjoint sets which have no common element with CARDINALITY. Elements of these sets are identifiers representing names of concepts. We call an element of ATT, ENT, REL, and LAB an *attribute name*, an *entity type name*, *a relationship type name* and a *label* (or *role*) name, respectively.

Definition 1 (Entity-Relationship schema) Let $\mathcal{A}$, $\mathcal{E}$, $\mathcal{L}$, and $\mathcal{C}$ be finite and not empty subsets of ATT, ENT, LAB, and CARDINALITY, respectively; and let $\mathcal{R}$ be a finite (possibly empty) subset of REL.
We say $\mathcal{S} = (e\_att, k\_att, r\_ent, r\_att)$ is a *conceptual* ER-*schema* (or a schema) over $\mathcal{A}$, $\mathcal{E}$, $\mathcal{L}$, $\mathcal{R}$, and $\mathcal{C}$ iff $e\_att : \mathcal{A} \to \mathcal{E}$, $k\_att : \mathcal{A} \to \mathcal{E}$, $r\_att : \mathcal{A} \to \mathcal{R}$, and $r\_ent : \mathcal{R} \times \mathcal{E} \times \mathcal{L} \to \mathcal{C}$ are partial functions satisfying the following constraints:

1. e_att is surjective, and $k\_att \sqsubseteq e\_att$.
2. $\forall R \in \mathcal{R}\ \ \exists (R, E_1, l_1), (R, E_2, l_2) \in def(r\_ent)$ such that $E_1 \neq E_2$; and $((R, E, l_1) \in def(r\_ent)) \wedge ((R, E, l_2) \in def(r\_ent)) \Rightarrow (l_1 \neq l_2)$.
3. $e\_att \cap r\_att = \emptyset$ and $e\_att \cup r\_att = \mathcal{A}$. ■

To see the connection between this definition and above rules, view the sets $\mathcal{E}$, $\mathcal{R}$, $\mathcal{A}$, $\mathcal{L}$, and $\mathcal{C}$ as the sets of entity types, relationship types, attribute names, roles and (min, max)-cardinalities of an application, respectively. Then, the definition provides a model of the rules as follows:

– $e\_att(A) = E$ means that A is an attribute of entity type E, and $k\_att(A) = E$ means that A is a key attribute of E.

– $r\_ent(R, E, l) = c$ means that entity type E participates in relationship type R with cardinality c and role r.

– $r\_att(A) = R$ means that A is an attribute of relationship type R.

Then, conditions 1–3 correspond to rules 1–3, respectively.
The above description can be expressed alternatively as follows:

Fact 1 (schema as algebra) Any ER-schema $\mathcal{S} = (e\_att, k\_att, r\_ent, r\_att)$ can be viewed as a finite model of the following specification and vice-versa, provided that the carrier of CARD is a subset of CARDINALITY:

```
Spec ER_Sch
 Sorts: ENTITY, RELSHIP, ATTRIBUTE, LABEL, CARD
 Ops:
    E_Att : ATTRIBUTE → ENTITY (partial),
    K_Att : ATTRIBUTE → ENTITY (partial),
    R_Att : ATTRIBUTE → RELSHIP (partial)
    R_Ent : RELSHIP ENTITY LABEL → CARD (partial)
 Axs:
  E,E':ENT , R:RELSHIP, A:ATTRIBUTE, l,l': LABEL, c, c':CARD
   ∀E ∃A  E_Att(A) = E,  ∀A  K_Att(A) = E_Att(A),
   ∀R  ∃E,  ∃E',  ∃c,  ∃c',  ∃l,  ∃l'
     (R_Ent(R, E, l) = c) ∧ (R_Ent(R, E', l') = c') ∧ (E≠E'),
   (R_Ent(R, E, l) = c)∧ (R_Ent(R, E, l') = c') ⇒ l≠l',
   ¬( ∃A ∃E ∃E' ((E_Att(A) = E)∧(R_Att(A)=E')))
    ∀A ∃E ((E_Att(A) = E) ∨(R_Att(A) = E')).
```

Indeed, in a finite algebra the carrier of each sort is a finite set and each operation is associated with a finite function. To view the schema $\mathcal{S}$ over $\mathcal{A}$, $\mathcal{E}$, $\mathcal{L}$, $\mathcal{R}$ and $\mathcal{C}$ as a finite algebra of the above specification (and vice-versa), take $\mathcal{A}$ as the carrier of ATTRIBUTE, $\mathcal{E}$ as the carrier of ENTITY, associate the function e_att with E_Att, the function k_att with K_Att, and so on. Then axioms of ER_Sch become equivalent to constraints 1-3 of Definition 1. As we mentioned earlier, the satisfaction of such constraints for finite functions can be checked. ■

Note: The first two axioms correspond to rule 1, the third and fourth axioms to rule 2 and the two last axioms to rule 3. Some authors [5] allow attribute names to be overloaded in an ER-diagram. This excludes a part of rule 3. To do so in our formalism, we have to consider e_att, k_att and r_att as multi-valued functions (binary relations) and change Definition 1 and Fact 1 mutatis-mutandis. To this end, for any multi-valued function $f : X \to \mathcal{P}_f(Y)$ and any $P \subseteq X$, we define $def(f) = \{x \in X \mid f(x) \neq \emptyset\}$ and $f(P) = \bigcup_{x \in P} f(x)$. We delete from Definition 1 the constraint $e_att \cap r_att = \emptyset$. In this way our formalism stands for those multi-valued functions without any change.

4 Instances and Loose Semantics

A *base interpretation* of an ER-schema $\mathcal{S}$ consists of associating a set $[\![A]\!]$ with each attribute $A \in \mathcal{A}$. The set $[\![A]\!]$ is called the *domain* of A and often written as $dom(A)$. Usually $dom(A)$ is the set of concrete *values* of a type. This type is often a basic type like `int`, `string`, `boolean`. But in most general case, it can also be a record type or a collection type (`sets`, `bags`, `lists`). Every base interpretation of $\mathcal{S}$ can be extended in a "canonical" way in order to interpret entity and relationship types of $\mathcal{S}$ too. The extension is defined as follows:

– Each entity type E with attributes $\{A_1, \cdots, A_n\}$ is interpreted by the set $[\![E]\!]$ of all $\{A_1, \cdots, A_n\}$-tuples over $[\![D_i]\!] = dom(A_i)$ $(1 \leq i \leq n)$. We view such a tuple as a function $e : \{A_1, \cdots, A_n\} \rightarrow \bigcup_{1 \leq i \leq n} [\![D_i]\!]$ such that $e(A_i)$, denoted $e.A_i$, is an element of $[\![D_i]\!]$. Each element e of $[\![E]\!]$ is called an *entity* of E and each $e.A_i$ is an *attribute value* participating in e.

– Each relationship type R with attributes $\{A_1, \cdots, A_k\}$ and entity types $\{E_1, \cdots, E_n\}$ is interpreted by the set $[\![R]\!]$ of all $\{A_1, \cdots, A_k, E_1, \cdots, E_n\}$-tuples over $[\![D_i]\!] = dom(A_i)$ $(0 \leq i \leq k)$ and $[\![E_j]\!]$ $(1 \leq j \leq m)$. Thus $[\![R]\!]$ is the set of functions $r : \{A_1, \cdots, A_k, E_1, \cdots, E_m\} \rightarrow (\bigcup_{1 \leq i \leq k} [\![D_i]\!]) \cup (\bigcup_{1 \leq j \leq m} [\![E_j]\!])$ such that $r(A_i)$ $(1 \leq i \leq k)$ is in $[\![D_i]\!]$ and $r(E_j)$ $(1 \leq j \leq m)$ is in $[\![E_j]\!]$. Each element r of $[\![R]\!]$ is called a *relationship* of R. Each $r(A_i)$, denoted $r.A_i$, is an *attribute value* participating in r and each $r(E_j)$, denoted $r.E_j$, is an entity participating in r.

Note: The above interpretations of $[\![E]\!]$ and $[\![R]\!]$ take into account the fact that sets of attributes and entity types are unordered sets at the conceptual level. Traditionally those sets are implicitly regarded as ordered sets. In that case one can define $[\![E]\!]$ as $[\![D_1]\!] \times \cdots \times [\![D_n]\!]$, and $[\![R]\!]$ as $[\![D_1]\!] \times \cdots \times [\![D_k]\!] \times [\![E_1]\!] \times \cdots \times [\![E_m]\!]$. Then $e.A_i$ corresponds to usual i-th component of the tuple e.

Definition 2 (Instance) An *instance* $\mathcal{I}$ of schema $\mathcal{S}$ consists of:
– a base interpretation $[\![\ ]\!]$,
– for each $E \in \mathcal{E}$, a finite subset $[\![E]\!]_{\mathcal{I}}$ of $[\![E]\!]$, and
– for each $R \in \mathcal{R}$, a finite subset $[\![R]\!]_{\mathcal{I}}$ of $[\![R]\!]$. ■

Definition 3 (Constraint satisfaction and loose semantics) Let $\mathcal{I}$ be an instance of an ER-schema and *r-ent(R, E,l)* = *c*. We say that $\mathcal{I}$ *satisfies* the cardinality constraint c for R and E, denoted $R \models_{\mathcal{I}} (E, c)$, iff $|e_R| \in [\![c]\!]$, where $|e_R|$ is the number of elements $r \in [\![R]\!]_{\mathcal{I}}$ in which e participates.
We say that an instance $\mathcal{I}$ for ER-schema satisfies the key constraints for entity type E iff $K = k\_att(E)$ satisfies:
$\forall e \in [\![E]\!]_{\mathcal{I}}, \forall e' \in [\![E]\!]_{\mathcal{I}}, (\forall A \in K, e.A = e'.A) \Longrightarrow e = e'$.
A *valid* instance[2] of $\mathcal{S}$ is an instance satisfying all constraints of $\mathcal{S}$. The collection of all valid instances of $\mathcal{S}$ is called the *loose semantics* of $\mathcal{S}$. ■

It is clear that the empty instance is valid. A schema is called *invalid* if its loose semantics contains only the empty instance. Checking whenever an instance is valid is a hard problem in general setting. However, the presence of some structural configurations may make this checking easier [4, 8, 11].

5 The ER-Meta Model

In this section we shall introduce a particular schema `META_ER`, called *the meta-schema*, in a way that each valid instance of the meta-schema is a schema, and vice-versa. In fact, the meta-schema is obtained by organizing, as a schema, the meta-concepts used in Definition 1.

Meta schema: We consider the following finite sets:
meta_$\mathcal{A}$ = {`A_name, E_name, R_name, L_name, Card`}
meta_$\mathcal{E}$ = { `ENTITY, RELSHIP, ATTRIBUTE, LABEL`}
meta_$\mathcal{R}$ = {`E_Att, K_Att, R_Att, R_Ent`}, meta_$\mathcal{L}$ = {`blank`},
`meta_`$\mathcal{C}$ = {`(0, 1), (0, N), (1, N), (2, N)`}.

and we suppose that these sets are subsets of `ATT`, `ENT`, `REL`, `LAB`, and `CARDINALITY`, respectively [3]. Then, we define the functions `meta_e_att, meta_k_att, meta_r_att: meta_`$\mathcal{A} \rightarrow$ `meta_`$\mathcal{E}$ as follows:

```
meta_e_att(A) = meta_k_att(A) =
```

[2] Some authors say *satisfiable schema* or *consistent schema.*

[3] Our definition of schema uses a role for every entity type. In fact, however, roles are only needed when two entity types participate in the same relationship type. We consider `blank` as a non printable element of `LAB` meaning an unnecessary role.

$\{\mathtt{A\_name} \mapsto \mathtt{ATTRIBUTE},\ \mathtt{E\_name} \mapsto \mathtt{ENTITY},\ \mathtt{R\_name} \mapsto \mathtt{RELSHIP},$
$\mathtt{L\_name} \mapsto \mathtt{LABEL},\ \mathtt{Card} \mapsto \mathtt{undefined}\}$

$$\mathtt{meta\_r\_att}(A) = \begin{cases} \mathtt{R\_Ent} & \text{if } A = \mathtt{Card}, \\ \mathtt{undefined} & \mathtt{otherwise} \end{cases}$$

Now, we define the function $\mathtt{meta\_r\_ent}\colon \mathtt{meta\_}\mathcal{R} \times \mathtt{meta\_}\mathcal{E} \times \mathtt{meta\_}\mathcal{L} \to \mathtt{meta\_}\mathcal{E}$ by the following table:

meta_r_ent	meta_$\mathcal{R}$	meta_$\mathcal{E}$	meta_$\mathcal{L}$	meta_$\mathcal{C}$
	E_Att	ATTRIBUTE	blank	(0, 1)
	E_Att	ENTITY	blank	(1, N)
	K_Att	ATTRIBUTE	blank	(0, 1)
	K_Att	ENTITY	blank	(0, N)
	R_Att	ATTRIBUTE	blank	(0, 1)
	R_Att	RELSHIP	blank	(0, N)
	R_Ent	RELSHIP	blank	(2, N)
	R_Ent	LABEL	blank	(1, N)
	R_Ent	ENTITY	blank	(0, N)

Fact 2 META_ER = (meta_e_att, meta_k_att, meta_r_ent, meta_r_att) is an ER-schema over meta_$\mathcal{A}$, meta_$\mathcal{E}$, meta_$\mathcal{L}$, meta_$\mathcal{R}$ and meta_$\mathcal{C}$ that we call the *meta-schema* of ER-model. ■

Meta instances: We define a base interpretation of META_ER as follows:
$[\![\mathtt{A\_name}]\!] = \mathtt{ATT}$, $[\![\mathtt{E\_name}]\!] = \mathtt{ENT}$, $[\![\mathtt{R\_name}]\!] = \mathtt{REL}$,
$[\![\mathtt{L\_name}]\!] = \mathtt{LAB}$, $[\![\mathtt{Card}]\!] = \mathtt{CARDINALITY}$.

Fact 3 Every schema $\mathcal{S}$ defines a valid instance $\mathcal{I}_{\mathcal{S}}$ of META_ER. ■

In fact META_ER as defined above is a special finite algebra of the specification ER_Sch.

Corollary 1 There is a valid instance $\mathcal{M}$ of META_ER that defines the schema META_ER itself.

Indeed, META_ER is an ER-schema. Hence, it defines a valid instance $\mathcal{M}$ of META_ER. This instance is defined by $[\![\mathtt{ATTRIBUTE}]\!]_{\mathcal{M}} := \mathtt{meta\_}\mathcal{A}$, $[\![\mathtt{ENTITY}]\!]_{\mathcal{M}} := \mathtt{meta\_}\mathcal{E}$, $[\![\mathtt{E\_Att}]\!]_{\mathcal{M}} := \mathtt{meta\_e\_att}$, and so on. Clearly $\mathcal{M}$ satisfies all constraints of META_ER. We summarize the above corollary by saying:

The meta schema is a valid instance of itself.

Fact 4 Each valid instance $\mathcal{S}$ of META_ER defines a ER-schema $\mathcal{S}_{\mathcal{I}}$. ■

Following Fact 1, each schema can be viewed as a finite model of ER_Sch. Let $\mathcal{M}$od(ER_Sch) be the class of all finite models of ER_Sch and $\mathcal{L}$oose(META_ER) the class of loose semantics of META_ER.

Theorem 1 $\mathcal{L}$oose(META_ER) is isomorphic to $\mathcal{M}$od(ER_Sch).

Indeed, for any valid instance $\mathcal{I}$ of META_ER define $\Psi(\mathcal{I}) = \mathcal{S}_{\mathcal{I}}$, and for any element $\mathcal{S}$ of $\mathcal{M}$od(ER_Sch) define $\Phi(\mathcal{S}) = \mathcal{I}_{\mathcal{S}}$. Then it can be proved that Φ and Ψ are inverse of each other. In particular, $\Psi(\Phi(\mathtt{META\_ER})) = \mathtt{META\_ER}$. ■

6 Related Works, Conclusion and further Research

Formalization and unification of conceptual modeling approaches have been of interest for many authors. Several works extend the original Chen proposal [3] to new concepts, including inheritance and objects [6, 5]. For instance, in [10] a formal higher order ER-model has been proposed. In this model the notion of relationship has been extended by introducing a relationship of higher order, permitting to nest relationships. To capture more semantics, a wide variety of constraints have been introduced in [8, 2, 11]. In [7, 11] a formal approach has been proposed for unification of these constraints. An amount of papers are devoted to the delicate problem of checking validity of an ER-schema in presence of constraints [4, 8, 7]. However, in all these works constraints are specified semantically. Meta modeling is another subject which has been used in various approaches of conceptual modeling, including reverse engineering, schema integration and model transformation. For instance, in [1] a meta model is used for reverse engineering, more precisely for discovering inheritance hidden links in a relational schema. From our point of view, the border between conceptual and physical levels is blurred in the above proposals, and their meta models are ad hoc and lack a formal basis. In this paper we have proposed a formal approach for ER-models. The approach does a neat separation between the specification of conceptual and physical data of an application. Another particularity of the present work is an attempt to distinguish between specification of constraints and their satisfaction. We have proved that our formalism is self-contained in the sense that all schemas are instances of a special schema: the meta schema. Proofs of main results are omitted for space limitation.

Many interesting aspects of conceptual modeling are not developed in this extended abstract, dynamic aspects and data warehousing [9] are among them. It is interesting to give a formal specification of the underlying dynamic system. The technique presented in this paper seems to be applicable for more sophisticated modeling approaches, especially for object modeling and a semantics of UML. We are currently investigating these research directions.

References

1. J. Akoka, I. Comyn-Wattiau, and N. Lammari. Relational Database Reverse Engineering, Elicitation of Generalization Hierarchies. *Advance in Conceptual Modeling, ER'99 Workshops on Evolution and Change in Data Management, LNCS 1727*, pages 173-185, November 1999.
2. J. B. Behm and T.J. Teorey. Relative Constraint in ER Data Model. *ER'93:Entitiy-Relationship Approach, 12th International Conference on the Entity-Relationship approach, LNCS 823*, pages 46-59, December 1993.
3. P.P. Chen. The Entity-Relationship Model — Towards a Unified view of Data. *ACM Transaction On Database System*, 1(1):9-36, March 1976.
4. J. Dullea and Il-Yeol Song. A Taxonomy of Recursive Relationships and Their Structural Validity in ER Modeling. *ER'99: Conceptual Modeling, 18th International Conference on Conceptual Modeling, LNCS 1728*, pages 384-398, 1999.
5. R. Elmasri and S. B. Navathe, Fundamentals of Database Systems, *The Bejamin Cummings Publishing Company, Inc., Second Edition*, 1994.
6. G. Engels, M. Gogolla, U. Hohenstein, K. Hulsmann, P. Lohr-Richter, G. Saake and H. D. Ehrich. Conceptual modelling of database applications using an extended ER model. *Data and Knowledge Engineering*, 9:157-204, 1993.
7. S. Hartmann. On the Consistency Of Int-cardinality Constraints. *ER'98: Conceptual Modeling, 17th International Conference on Conceptual Modeling, LNCS 1507*, Pages 150-163, November 1998.
8. M. Lenzerini and P. Nobili. On the satisfiability of dependency constraints in Entity-Relationship schemata. *Information Systems*, 15(4):453-461, 1990.
9. D. Moody and Kortink. From Entities to Stars, Snowflakes, Constellations and Galixies: A Methodology for Data Warehouse Design. *18th International Conference on Conceptual Modelling Proceedings*, pages 114-130, March 1999.
10. B. Thalheim. The Higher-Order Entity-Relationship Model and DB2. *MFDBS'89: 2nd Symposium on Mathematical Fundamentals of Database Systems, LNCS 364*, pages 382-397, June 26-30 1989.
11. B. Thalheim. Fundamentals of Cardinality Constraints. *ER'92: Entity-Relationship Approach, 11th International Conference on the Entity-Relationship Approach, LNCS 645*, pages 7-23, October 7-9 1992.

Integrating and Managing Conflicting Data

Sergio Greco, Luigi Pontieri and Ester Zumpano

DEIS, Università della Calabria
87030 Rende, Italy
e-mail: {greco,pontieri,zumpano}@si.deis.unical.it

1 Introduction

The aim of data integration is to provide a uniform integrated access to multiple heterogeneous information sources, which were designed independently for autonomous applications and whose contents are strictly related. The heterogeneity among sources may range from hardware and software platform to the data model and the schema used to represent information. The problem of integrating heterogeneous sources has been deeply investigated in the fields of multidatabase systems [4], federated databases [16] and, more recently, mediated systems [14, 16]. Mediator-based architectures are characterized by the presence of two types of components: *wrappers*, which translate the local languages, models and concepts of the data sources into the global ones, and *mediators*, which take in input information from one or more components below them and provide an integrated view of it [12, 7]. Views, managed by mediators, may be virtual or materialized. When a mediator receives a query, it dispatches subqueries to the components below it (wrappers and/or mediators), collects the results and merges them in order to construct the global answer. Mediators have to cope with schema and value inconsistencies that may be present in the information coming from the different sources. The first kind of inconsistency arises when different sources use different schemas to model the same concept, while the second one arises when different sources record different values for the same object [11, 15].

In this paper, we focus our attention on the integration of conflicting instances [1, 2, 6] related to the same concept and possibly coming from different sources. We introduce an operator, called *Merge operator*, which allows us to combine data coming from different sources, preserving the information contained in each of them. Generally, at any level of the architecture, the integrated information may not satisfy some integrity constraints associated with the schema of the mediator. We introduce a variant of the merge operator, i.e. the *Prioritized Merge operator*, which can be employed to combine data using preference criteria and present a technique which permits us to compute consistent answers, i.e. maximal sets of atoms which do not violate the constraints. This technique is based on the identification of tuples satisfying integrity constraints and on the selection of tuples satisfying the query.

2 Data Integration

Mediators provide an integrated view of a set of information sources. Each of these sources may be a source database or a database view (virtual or materialized) furnished by another mediator.

At each level of the integration system, the information provided by different sources and related to the same concept is combined. The necessity of completing the information regarding a concept is due to the fact that some information may not be available at a source because it is not modeled within the schema of the source or simply because some data instances contain undefined values for some attributes. The way we integrate different sources preserves the information contained in each of them, since we try to complete the information but we never modify that already available.

Let us introduce some basic definitions in order to simplify the description of our approach. A mediator has its own schema, that we call *mediator schema*, and a set of integrity constraints whose satisfaction means that data are consistent. The mediator schema represents, in an integrated way, some relevant concepts that may be modeled differently within the schemas of different sources. Integrity constraints are first order formulas which must always be true. Although in this paper we only consider functional dependencies, our approach to managing inconsistent data is more general.

Let us adopt the relational model for referring schemas and instances pertaining to the mediator and the sources it integrates.

Notation: Let R be a relation name, then we denote by:

- *attr(R)* the set of attributes of R;
- *key(R)* the set of attributes in the primary key of R;
- *inst(R)* an instance of R (set of tuples).

Moreover, given a tuple $t \in inst(R)$, $key(t)$ denotes the values of the key attributes of t.

We assume that relations associated with the same concept have been homogenized with respect to a common ontology, so that attributes denoting the same concepts have the same name [15]. We say that two homogenized relations R and S, associated with the same concept, are *overlapping* if $key(R) = key(S)$.

Definition 1. Given a set of attributes DS and two tuples t_1, t_2 over DS. We say that $t_1 \sqsubseteq t_2$ if for each attribute A in DS, $t_1[A] = t_2[A]$ or $t_1[A] = null$. Moreover, given two relations R and S over DS, $R \sqsubseteq S$ if $\forall t_1 \in R\ \exists t_2 \in S$ s.t. $t_1 \sqsubseteq t_2$. □

Definition 2. Let $R_1, ..., R_n$ be a set of overlapping relations. A relation R is a *super relation* of $R_1, ..., R_n$ if the following conditions hold:

- $attr(R) = \bigcup_{i=1}^{n} attr(R_i)$,
- $inst(R_i) \sqsubseteq \pi_{attr(R_i)} inst(R)$,
- $key(R) = key(R_i) \quad \forall\ i = 1..n$. □

Moreover, if R is a super relation of $R_1, ..., R_n$, then we say that R_i is a *sub-relation* of R for $i = 1..n$.

A mediator has to define the content of any global relation as an integrated view of the information provided by all its sub-relations. Once the logical conflicts due to the schema heterogeneity have been resolved, conflicts may arise, during the integration process, among instances provided by different sources. In particular, the same real-world object may correspond to many tuples (possibly residing in different overlapping relations), that may have the same value for the key attributes but different values for some non-key attribute.

A set of tuples with the same value for the key attributes is called *c-tuple* (cluster of tuples) [1]. In this context a relation may be seen as a set of c-tuples.

An important feature of the integration process is related to the way conflicting tuples provided by overlapping relations are combined. In the following section we define an operator, which allows us to integrate a set of relations, preserving the original information.

The Merge Operator

Let us first introduce the binary operator Θ which replaces null values occurring in a relation with values taken from a second one. In more detail, given two relations R and S such that $attr(S) \subseteq attr(R)$, the operator is defined as follows:

$$\Theta(R,S) = \{\ t \in R \mid \nexists t_1 \in S\ s.t.\ key(t) = key(t_1)\ \} \cup$$
$$\left\{ t \mid \exists t_1 \in R, \exists t_2 \in S\ s.t.\ \forall a \in attr(R)\ \left(t[a] = \begin{cases} t_2[a] & \text{if } (a \in attr(S) \wedge t_1[a] = \perp) \\ t_1[a] & \text{otherwise} \end{cases} \right) \right\}$$

Given two overlapping relations S_1 and S_2, the *merge operator*, denoted by $\boxtimes$, integrates the information provided by S_1 and S_2. Let $S = S_1 \boxtimes S_2$, then the schema of S contains both the attributes in S_1 and S_2, and its instance is obtained by completing the information coming from each input relation with that coming from the other one.

Definition 3. Let S_1 and S_2 be two overlapping relations. The *merge operator* is a binary operator defined as follows:

$$S_1 \boxtimes S_2 = \Theta(S_1 \sqsupset\!\bowtie S_2, S_2) \cup \Theta(S_1 \bowtie\!\sqsubset S_2, S_1)$$

□

$S_1 \boxtimes S_2$ computes the full outer join and extends tuples coming from S_1 (resp. S_2) with the values of tuples of S_2 (resp. S_1) having the same key. The extension of a tuple is carried out by the operator Θ which replaces null values appearing in a given tuple of the first relation with values appearing in some correlated tuple of the second relation. Thus, the merge operator applied to two relation S_1 and S_2 'extends' the content of tuples in both S_1 and S_2.

Proposition 1. *Let S_1 and S_2 be two overlapping relations, then:*

- $attr(S_1 \boxtimes S_2) = attr(S_1) \bigcup attr(S_2)$
- $key(S_1 \boxtimes S_2) = key(S_1) = key(S_2)$,
- $inst(S_1) \sqsubseteq \pi_{attr(S_1)} inst(S_1 \boxtimes S_2)$ *and* $inst(S_2) \sqsubseteq \pi_{attr(S_2)} inst(S_1 \boxtimes S_2)$. □

Example 1. Consider the relations $S1$ and $S2$ reported in Fig. 1 in which K is the key of the relations and the functional dependency $Title \rightarrow Author$ holds. The relation T is obtained from the merging of S_1 and S_2, i.e. $T = S_1 \boxtimes S_2$.

K	*Title*	*Author*
1	*Moon*	*Greg*
2	*Money*	*Jones*
3	*Sky*	*Jones*

S_1

K	*Title*	*Author*	*Year*
3	*Flowers*	*Smith*	1965
4	*Sea*	*Taylor*	1971
7	*Sun*	*Steven*	1980

S_2

K	*Title*	*Author*	*Year*
1	*Moon*	*Greg*	⊥
2	*Money*	*Jones*	⊥
3	*Sky*	*Jones*	1965
3	*Flowers*	*Smith*	1965
4	*Sea*	*Taylor*	1971
7	*Sun*	*Steven*	1980

T

Fig. 1.

Let S_1 and S_2 be two overlapping relations, let $K = key(S_1) = key(S_2)$, $A = \{a_1, ..., a_n\} = attr(S_1) \cap attr(S_2) - K$, $B = \{b_1, ..., b_m\} = attr(S_1) - attr(S_2)$ and $C = \{c_1, ..., c_q\} = attr(S_2) - attr(S_1)$. The merge operator introduced in Definition 3 can easily be expressed by means of the following SQL statement (where, given a relation R and a set of attributes $X = X_1, ..., X_t$, the notation $R.X$ stands for $R.X_1, ..., R.X_t$):

```
SELECT  S1.K, S1.B, COALESCE(S1.a1, S2.a1), .., COALESCE(S1.an, S2.an), S2.C
FROM    S1 LEFT OUTER JOIN S2 ON S1.K = S2.K
UNION
SELECT  S2.K, S1.B, COALESCE(S2.a1, S1.a1), .., COALESCE(S2.an, S1.an), S2.C
FROM    S1 RIGHT OUTER JOIN S2 ON S1.K = S2.K
```

where the standard operator COALESCE($a_1, ..., a_n$) returns the first not null value in the sequence.

Proposition 2.

- $S_1 \boxtimes S_2 = S_2 \boxtimes S_1$ *(commutative property),*
- $(S_1 \boxtimes S_2) \boxtimes S_3 = S_1 \boxtimes (S_2 \boxtimes S_3)$ *(associative property),*
- $S_1 \boxtimes S_1 = S_1$ *(idempotent property).* □

Obviously, given a set of overlapping relations $S_1, S_2, ..., S_n$, the associated super-relation S can be obtained as $S = S_1 \boxtimes S_2 \boxtimes ... \boxtimes S_n$. In other words S is the integrated view of $S_1, S_2, ..., S_n$.

The problem we are considering is similar to the one treated in [15], which assumes the source relations involved in the integration process have previously been homogenized. In particular, any homogenized source relation is a fragment of the global relation, that is it contains a subset of the attributes of the global relation and has the same key K. The technique proposed in [15] makes use of an operator $\bowtie$, called *Match Join*, to manufacture tuples in global relations using fragments. This operator consists of the outer-join of the $ValSet$ of each attribute, where the $ValSet$ of an attribute A is the union of the projections of each fragment on $\{K, A\}$. Therefore, the application of the Match Join operator produces tuples containing associations of values that may not be present in any fragment.

Example 2. We report below the relation obtained by applying the Match Join operator to the relations S_1 and S_2 of Example 1.

K	*Title*	*Author*	*Year*
1	*Moon*	*Greg*	$\perp$
2	*Money*	*Jones*	$\perp$
3	*Sky*	*Jones*	1965
3	*Sky*	*Smith*	1965
3	*Flowers*	*Smith*	1965
3	*Flowers*	*Jones*	1965
4	*Sea*	*Taylor*	1971
7	*Sun*	*Steven*	1980

T

The Match Join operator applied to the source relations of Example 1 produces tuples violating the functional dependency *Title* → *Author* since it mix values coming from different tuples with the same key in all possible ways. The merge operator introduced here only tries to derive unknown values and the derived relation satisfies the functional dependencies for each cluster of tuples. Observe also that the match join operator does not satisfy the idempotent property, i.e. $S_1 \overline{\bowtie} S_1 \neq S_1$.

3 Answering Queries Satisfying User Preferences

In this section we introduce a variant of the merge operator, which allows the mediator to answer queries according to the user preferences. Preference criteria are expressed by a set of constraints called *preference constraints* which permit us to define a partial order on the source relations.

A *preference constraint* is a rule of the form $S_i \ll S_j$, where S_i, S_j are two source relations. Preference constraints imply a partial order on the source relations. We shall write $S_1 \ll S_2 \ll ... \ll S_k$ as a shorthand of $\{S_1 \ll S_2, S_2 \ll S_3, ..., S_{k-1} \ll S_k\}$. The presence of such constraints requires the satisfaction of preference criteria during the computation of the answer. A priority statement of the form $S_i \ll S_j$ specifies a preference on the tuples provided by the relation S_i with respect to the ones provided by the relation S_j.

The Prioritized Merge Operator

In order to satisfy preference contraints, we introduce an asymmetric merge operator, called *prioritized merge operator*, which gives preference to data coming from the left relation when conflicting tuples are detected.

Definition 4. Let S_1 and S_2 be two overlapping relations and $S_2' = S_2 \ltimes (\pi_{key(S_2)} S_2 - \pi_{key(S_1)} S_1)$ the set of tuples in S_2 not joining with any tuple in S_1. The *prioritized merge operator* is defined as follows:

$$S_1 \lhd S_2 = \Theta(S_1 \sqsupset\!\bowtie S_2, S_2) \cup (S_1 \bowtie\!\sqsubset S_2') \qquad \Box$$

The prioritized merge operator includes all tuples of the left relation and only the tuples of the right relation whose key does not identify any tuple in the left relation. Moreover, only tuples 'coming' from the left relation are extended since tuples coming from the right relation, joining some tuples coming from the left relation, are not included. Thus, when integrating relations conflicting on the key attributes, the prioritized merge operator gives preference to the tuples of the left side relation and completes them with values taken from the right side relation.

Example 3. Consider the source relations S_1 and S_2 of Example 1, The relation $T = S_1 \lhd S_2$ is:

K	*Title*	*Author*	*Year*
1	*Moon*	*Greg*	$\perp$
2	*Money*	*Jones*	$\perp$
3	*Sky*	*Jones*	1965
4	*Sea*	*Taylor*	1971
7	*Sun*	*Steven*	1980

T

The merged relation obtained in this case differs from that of Example 1 because it does not contain the tuple $(3, Flowers, Smith, 1965)$ coming from relation S_2.

Proposition 3. *Let S_1 and S_2 be two relations, then:*

- $S_1 \lhd S_2 \subseteq S_1 \boxtimes S_2$,
- $S_1 \boxtimes S_2 = (S_1 \lhd S_2) \cup (S_2 \lhd S_1)$,
- $S_1 \lhd S_1 = S_1$. □

The merge operation $S_1 \lhd S_2$, introduced in Definition 4 can easily be expressed by means of an SQL statement, as follows:

```
SELECT S1.K, S1.B, COALESCE(S1.a1, S2.a1), .., COALESCE(S1.an, S2.an), S2.C
FROM   S1 LEFT OUTER JOIN S2 ON S1.K = S2.K
UNION
SELECT S2.K, NULL(B), S2.A, S2.C
FROM   S2, S1
WHERE S2.K NOT IN (SELECT S1.K FROM S1)
```

where the function NULL(B) assigns null values to the attributes in B.

4 Managing Inconsistent data

We assume that each mediator component involved in the integration process contains an explicit representation of intentional knowledge, expressed by means of integrity constraints. Integrity constraints, usually defined by first order formulas or by means of special notations, express semantic information about data, i.e. relationships that must hold among data. Generally, a database D has a set of integrity constraints $\mathcal{IC}$ associated with it. D is said to be *consistent* if $D \models \mathcal{IC}$, otherwise it is inconsistent. In this paper we concentrate on functional dependencies. We present a technique which permits us to compute consistent answers for possibly inconsistent databases. The technique is based on the generation of a disjunctive program $\mathcal{DP}(\mathcal{IC})$ derived from the set of integrity constraints $\mathcal{IC}$. Before introducing our technique we briefly recall the definition of disjunctive program.

An *extended Datalog* program is a set of rules of the form

$$A_0 \vee ... \vee A_k \leftarrow B_1, ..., B_m, not\ B_{m+1}, ..., not\ B_n \qquad k+n>0$$

where $A_0, ..., A_k, B_1, ..., B_n$ are extended atoms. A generalized disjunctive program may also contain disjunctions in the body of rules, that is each B_i is a disjunction of literals. The semantics of an extended program $\mathcal{P}$ is defined by its minimal models (denoted by $MM(\mathcal{P})$) by considering each negated predicate symbol, say $\neg p$, as a new symbol syntactically different from p and by adding to the program, for each predicate symbol p with arity n the constraint $\leftarrow p(X_1, ..., X_n), \neg p(X_1, ..., X_n)$. Thus, the semantics of the program $P = \{\neg a \vee b \leftarrow\}$ is given by the two models $\{b\}$ and $\{\neg a\}$, and, therefore, a is false in all models.

The computation of the consistent answers of a query G can be derived by considering the minimal models of the program $\mathcal{DP}(\mathcal{IC})$ over the database D.

Definition 5. Let c be a functional dependency $x \rightarrow y$ over P, which can be expressed by a formula of the form $(\forall x, y, z, u, v)[\,P(x, y, u) \wedge P(x, z, v) \supset y = z\,]$ then, $dj(c)$ denotes the extended disjunctive rule

$$\neg P(x, y, u) \vee \neg P(x, z, v) \leftarrow P(x, y, u),\ P(x, z, v), y \neq z$$

Let $\mathcal{IC}$ be a set of functional dependencies, then $\mathcal{DP}(\mathcal{IC}) = \{\ dj(c) \mid c \in \mathcal{IC}\ \}$. □

Thus, $\mathcal{DP}(\mathcal{IC})$ denotes the set of disjunctive rules obtained by rewriting $\mathcal{IC}$. MM$(\mathcal{DP}(\mathcal{IC}), D)$ denotes the set of minimal models of $\mathcal{DP}(\mathcal{IC}) \cup D$.

Definition 6. Given a database D, a set of integrity constraints $\mathcal{IC}$ and a query G. Then, the consistent answer of G over D consists of the three distinct sets denoting, respectively, true, undefined and false atoms:

- $Ans^+(G, D, \mathcal{IC}) = \{\ q(t) \in D \mid\ \nexists M \in \mathrm{MM}(\mathcal{DP}(\mathcal{IC}), D)\ s.t.\ \neg q(t) \in M\ \}$
- $Ans^u(G, D, \mathcal{FD}) = \{\ q(t) \in D \mid \exists M_1, M_2 \in \mathrm{MM}(\mathcal{DP}(\mathcal{IC}), D)\ s.t.\ \neg q(t) \in M_1\ and\ \neg q(t) \notin M_2\ \}$
- $Ans^-(G, D, \mathcal{FD})$ denotes the set of atoms which are neither true nor undefined (false atoms). □

Theorem 1. *Let D be an integrated database, $\mathcal{FD}$ a set of functional dependencies and G a query. Then, the computation of a consistent answer of G over D can be done in polynomial time.* □

Example 4. Consider the integrated relation T of Example 1 and the functional dependency

$$K \rightarrow (Title, Author, Year)$$

stating that K is a key for the relation. The functional dependency can be rewritten as first order formulas:

$$(\forall x, y, z, w, y', z', w')[T(x, y, z, w), T(x, y', z', w') \supseteq y = y', z = z', w = w']$$

The associated disjunctive program is

$$\neg T(x, y, z, w) \vee \neg T(x, y', z', w') \leftarrow T(x, y, z, w), T(x, a, b, c), (y \neq a \vee z \neq b \vee w \neq c)$$

The above program has two stable models

$$M_1 = D \cup \{\neg T(3, Sky, Jones, 1965)\}$$

and

$$M_2 = D \cup \{\neg T(3, Flowers, Smith, 1965)\}.$$

Thus, the answer to the query asking for the title of the book with code 2 is $Money$ whereas the answer to the query asking for the title of the book with code 3 is unknown since there are two alternative values. □

References

1. S. Argaval, A.M. Keller, G.Wiederhold, and K. Saraswat. Flexible Relation: an Approach for Integrating Data from Multiple, Possibly Inconsistent Databases. In IEEE *Int. Conf. on Data Engineering*,1995.
2. M. Arenas, L. Bertossi, J. Chomicki, Consistent Query Answers in Inconsistent Databases. *Proc. PODS 1999*, pp. 68–79, 1999.
3. C. Baral, S. Kraus, J. Minker, Combining Multiple Knowledge Bases. *IEEE-Trans. on Knowledge and Data Engineering*, 3(2): 208-220 (1991)
4. Y. Breitbart, Multidatabase interoperability. *Sigmod Record 19(3)* (1990), 53–60.
5. F. Bry, Query Answering in Information System with Integrity Constraints, In *IFIP WG 11.5 Working Conf. on Integrity and Control in Inform. System*, 1997.
6. P. M. Dung, Integrating Data from Possibly Inconsistent Databases. *Proc. Int. Conf. on Cooperative Information Systems*, 1996: 58-65
7. H. Garcia-Molina, Y. Papakonstantinou, D. Quass,A. Rajaraman , Y. Sagiv, J. Ullman, V. Vassalos, J. Widom. The TSIMMIS approach to mediation: Data models and languages. *Journal of Intelligent Information Systems 8* (1997), 117–132.
8. M. Gelfond,V. Lifschitz(1991), Classical Negation in Logic Programs and Disjunctive Databases, *New Generation Computing, 9*, 365–385.
9. J. Grant, V. S. Subrahmanian: Reasoning in Inconsistent Knowledge Bases. *IEEE-Trans. on Knowledge and Data Eng.*, 7(1): 177-189, 1995
10. S. Greco, E. Zumpano Querying Inconsistent Database *Logic for Programming and Automated Reasoning*, pages 308-325, 2000.
11. R. Hull, Managing Semantic Heterogeneity in Databases: a Theoretical Perspective, *Proc. Symposium on Principles of Database Systems*, 1997: 51-61
12. A. Levy, A. Rajaraman, J. Ordille, Querying heterogeneous information sources using source descriptions. *Proc. VLDB Conf.*, 1996, pages 251–262.
13. J. Lin, A. O. Mendelzon, Knowledge Base Merging by Majority, in R. Pareschi and B. Fronhoefer (eds.), *Dynamic Worlds*, Kluwer, 1999. Kluwer, 1999.
14. J. D. Ullman, Information Integration Using Logical Views, *Theoretical Computer Science*, 239(2) 2000: 189-210
15. L.L. Yan, M. T. Ozsu, Conflict Tolerant Queries in Aurora *Proc. Coopis Conf.*, 1999: 279-290
16. G. Wiederhold, Mediators in the architecture of future information systems. *IEEE Computer 25(3)* (1992), 38–49.

A Knowledge Engineering Approach to Deal with 'Narrative' Multimedia Documents

Gian Piero Zarri

Centre National de la Recherche Scientifique (CNRS),
44 rue de l'Amiral Mouchez,
F-75014 Paris, France
zarri@ivry.cnrs.fr

Abstract. I describe here NKRL (Narrative Knowledge Representation Language), a modelling formalism used to deal with narrative multimedia documents. In these documents, the main part of the information content concerns the description of 'events' that relate the real or intended behaviour of some 'characters'. Narrative documents of an industrial and economic interest correspond, e.g., to news stories, corporate documents, normative and legal texts, intelligence messages, medical records, etc.

1 Introduction

Narrative documents, or 'narratives', are multimedia documents that describe the actual (or intended) state or behaviour of some 'actors' (or 'characters', 'personages' etc.). These try to attain a specific result, experience particular situations, manipulate some (concrete or abstract) materials, communicate with other people, send or receive messages, buy, sell, deliver etc. Leaving pure fiction aside, we can note that:

- A considerable amount of the natural language (NL) information that is relevant from an economic point of view deals, in reality, with narratives. This is true, of course, for the news story documents, but also for corporate information (memos, policy statements, reports, minutes etc.), intelligence messages, medical records etc., to say nothing of notarised deeds, sentences and other legal documents.
- In the narrative documents, the actors or personages are not necessarily human beings. We can have narrative documents concerning, e.g., the vicissitudes in the journey of a nuclear submarine (the 'actor', 'subject' or 'character') or the various avatars in the life of a commercial product. This personification process can be executed to a very large extent, giving then rise to narrative documents apparently very removed from any human context.
- It is not even necessary that the narrative situations be recorded in NL documents. Let us consider a collection of Web images, where one represents an information that, verbalised, could be expressed as "Three nice girls are lying on the beach". Having at our disposals tools — like those described in this paper — for coding the 'meaning' of generic narrative documents in a machine-understandable way, we can directly 'annotate' the picture using this code and without any recourse to a previous NL rendering. The same is, obviously, possible for 'narrative' situations described in video or digital audio documents.

In this paper, I will describe the use of NKRL (Narrative Knowledge Representation Language), see [1, 2], to represent the gist of economically relevant narratives. Note that NKRL has been used as 'the' modelling (knowledge representation) language for narratives in European projects like Nomos (Esprit P5330), Cobalt (LRE P61011) WebLearning (GALILEO Actions), Concerto (Esprit P29159) and in the Euforbia project (IAP P26505) actually under way.

2 General Information about NKRL

Traditionally, the NKRL knowledge representation tools are presented as organised into four connected 'components', the definitional, enumerative, descriptive and factual component.

The 'definitional component' supplies the tools for representing the 'concepts', intended here as the 'important notions' that we must take into account in a given application domain. In NKRL, a concept is represented, substantially, as a frame-like data structure, i.e., as an n-ary association of triples 'name-attribute-value' that have a common 'name' element. This name corresponds to a symbolic label like `human_being`, `taxi_` (the general class referring to all the possible taxis, not a specific cab), `city_`, `chair_`, `gold_`, etc. NKRL concepts are inserted into a generalisation/specialisation hierarchy that, for historical reasons, is called H_CLASS(es), and which corresponds to the usual ontologies of terms see, e.g. [3].

The 'enumerative component' of NKRL concerns the tools for the formal representation, as (at least partially) instantiated frames, of the concrete realisations (lucy_, taxi_53, chair_27, paris_) of the H_CLASS concepts. In NKRL, the instances of concepts take the name of *individuals*. Individuals are then countable and, like the concepts, possess unique symbolic labels (lucy_ etc.). Throughout this paper, I will use the italic type style to represent a *concept_*, the roman style to represent an individual_.

The 'descriptive' and 'factual' tools concern the representation of the 'events' proper to a given domain — i.e., the coding of the *interactions* among the *particular concepts and individuals* that *play a role* in the contest of these events.

The descriptive component concerns the modelling tools used to produce the formal representations (called 'templates') of some *general narrative classes*, like "moving a generic object", "formulate a need", "having a negative attitude towards someone", "be present somewhere". In contrast to the traditional frame structures used for concepts and individuals, templates are characterised by the association of quadruples connecting together the *symbolic name* of the template, a *predicate* and the *arguments* of the predicate introduced by named relations, the *roles*. The quadruples have in common the 'name' and 'predicate' elements. If we denote then with L_i the generic symbolic label identifying a given template, with P_j the predicate used in the template, with R_k the generic role and with a_k the corresponding argument, the NKRL data structures for templates have the following general format:

$$(L_i\ (P_j\ (R_1\ a_1)\ (R_2\ a_2)\ \ldots\ (R_n\ a_n))), \qquad (1)$$

see the example in Figure 1, commented below. Presently, the predicates pertain to the set {BEHAVE, EXIST, EXPERIENCE, MOVE, OWN, PRODUCE, RECEIVE}, and the roles to the set {SUBJ(ect), OBJ(ect), SOURCE, BEN(e)F(iciary), MODAL(ity), TOPIC, CONTEXT}. Templates are structured into an inheritance hierarchy, H_TEMP(lates), which corresponds, therefore, to a new sort of ontology, an 'ontology of events'.

The instances (called 'predicative occurrences') of the templates, i.e., the representation of single, specific elementary events — see examples like "Tomorrow, I will move the wardrobe" or "Lucy was looking for a taxi" — are, eventually, in the domain of the last component, the factual one.

3 A Simple Example

To represent an elementary event like "On April 5th, 1982, Gordon Pym is appointed Foreign Secretary by Margaret Thatcher", we must select firstly the template corresponding to 'nominate to a post', which is represented in the upper part of Figure 1. We can note an important point. Unlike, e.g., canonical graphs in Sowa's conceptual graphs theory, [4], which must be explicitly defined for each new application, the (about 200) templates making up actually the H_TEMP hierarchy are fixed and fully defined. We talk, sometimes, about the 'catalogue' of the NKRL templates, and we say that they are part and parcel of the definition of the language. Moreover, when needed, it is easy to derive from the existing templates new templates that are needed for a particular application. If they prove to be sufficiently general, they are then added to the 'catalogue'. Therefore, H_TEMP is a continuously growing structure.

The 'position' code shows the place of this 'nomination' template within the OWN branch (5.) of the H_TEMP hierarchy: this template is then a specialisation (see the 'father' code) of the particular OWN template that corresponds to 'being in possession of a post'. The (mandatory) presence of a 'temporal modulator', 'begin' indicates that the only timestamp (t_1) which can be associated with the predicative occurrences derived from the 'nomination' template corresponds to the beginning of the state of being in possession — here, to the nomination. In the occurrences, time stamps are represented in general through two 'temporal attributes', date-1 and date-2, see [2]. In the occurrence c1 of Figure 1, this interval is reduced to a point on the time axis, as indicated by the single value, '5-april-1982' (the nomination date), associated with the attribute date-1.

The argument of the predicate (the a_k terms in formula (1) of the previous Section) are represented by variables with associated constraints; these last are expressed as concepts or combinations of concepts, i.e., making use of the terms of the H_CLASS hierarchy (definitional component). The 'location attributes', represented in the predicative occurrences as lists, are linked with the predicate arguments by using the colon operator, ':'. The constituents (SOURCE in Figure 1) included in square brackets are optional; the symbol '( )' means: forbidden for a given template.

The role fillers in the predicative occurrence represented in the lower part of Figure 1 conform to the constraints of the father-template. For example, gordon_pym is an individual (enumerative component) instance of the sortal concept *individual_person* which is, in turn, a specialisation of *human_being*; *foreign_secretary* is a specialisation of *post_*, etc. — note that the filler of a SOURCE role always represents the 'originating factor' of the event.

```
name: Own:NamingToPost
father: Own:BeingInPossessionOfPost
position: 5.1221
NL description: 'A Human Being is Appointed to a Post'

OWN SUBJ             var1: [var2]
    OBJ              var3
    [SOURCE var4: [var5]]
    (BENF)*
    [MODAL           var6]
    [TOPIC           var7]
    [CONTEXT         var8]
    { [ modulators ], begin }

    var1   =       <human_being>
    var3   =       <post_>
    var4   =       <human_being_or_social_body>
    var6   =       <action_name>
    var7   =       <property_>
    var8   =       <event_> | <action_name>
    var2, var5 =   <physical_location>

c1)  OWN     SUBJ     gordon_pym
             OBJ      foreign_secretary
             SOURCE   margaret_thatcher
             [begin]
             date-1:  (5-april-1982)
             date-2:
```

Fig. 1. Deriving a predicative occurrence from a template

4 Additional Properties of the NKRL Language

The basic NKRL tools are enhanced by two additional classes of formalisms:

- the AECS 'sub-language', see [1], that allows the construction of complex (structured) predicate arguments, called 'expansions';
- the second order tools (binding structures and completive construction), see [2], used to code the 'connectivity phenomena' (logico-semantic links) that, in a narrative situation, can exist between single narrative fragments (corresponding to single NKRL predicative structures).

Figure 2 translates the news story: "This morning, the spokesman said in a newspaper interview that, yesterday, his company has bought three factories abroad".

```
c2) MOVE  SUBJ     (SPECIF human_being_1 (SPECIF spokesman_
                                                  company_1))
          OBJ      #c3
                BENF   newspaper_1
                MODAL          interview_
                date-1:  today_
                date-2:

c3) PRODUCE SUBJ     company_1
            OBJ      (SPECIF purchase_1 (SPECIF factory_99
                          (SPECIF cardinality_ 3))): (abroad_)
            date-1: yesterday_
            date-2:

    [ factory_99
            InstanceOf : factory_
            HasMember    : 3 ]
```

Fig. 2. An example of completive construction

today_ and yesterday_ are two fictitious individuals introduced here, for simplicity's sake, in place of real or approximate dates, see [2]. The 'attributive operator', SPECIF(ication), which appears in both the occurrences c2 and c3, is one of the four operators that make up the AECS sub-language. AECS includes the disjunctive operator

(ALTERNative = A), the distributive operator (ENUMeration = E), the collective operator (COORDination = C), and the attributive operator (SPECIFication = S). Informally, the semantics of SPECIF can be explained in this way: the SPECIF lists, with syntax (SPECIF e_i p_1 ... p_n), are used to represent some of the properties p_i that can be asserted about the first argument e_i, concept or individual, of the operator, e.g., human_being_1 and *spokesman_* in the occurrence c2 of Figure 2.

In coding narrative information, one of the most difficult problems concerns finding a way of dealing with the 'connectivity phenomena' like causality, goal, indirect speech, co-ordination and subordination, etc. — in short, all those phenomena that, in a sequence of statements, cause the global meaning to go beyond the simple addition of the information conveyed by each single statement. In NKRL, the connectivity phenomena are dealt with making a (limited) use of second order structures: these are obtained from a *reification* of the predicative occurrences based on the use of their symbolic labels — like c2 and c3 in Figure 2. A simple example of second order structure is then given by the so-called 'completive construction', that consists in accepting as filler of a role in a predicative occurrence the symbolic label of another predicative occurrence. For example, see Figure 2, we can remark that the particular MOVE template (descriptive component) which is at the origin of c2 is systematically used to translate any sort of explicit or implicit transmission of an information ("The spokesman said..."). In this example of completive construction, the filler of the OBJ(ect) slot in the occurrence (c2) which materialises the 'transmission' template is a symbolic label (c3) that refers to another occurrence, i.e. the occurrence bearing the informational content to be spread out (" ...the company has bought three factories abroad"). Other, more complex ways of dealing with the connectivity phenomena are expounded, e.g., in [2].

5. The Query Language, and the FUM Module

Figure 3 shows the native NKRL coding (above) of an extremely simple narrative fragment: "On June 12, 1997, John and Peter were admitted (*together*) to hospital" — note that adding the indication 'together' forces the use of the AECS COORDination operator in the complex argument introduced by SUBJ.

```
c2)       EXIST  SUBJ   (COORD john_ peter_): (hospital_1)
          {begin}
          date-1: 2-june-1997
          date-2:

 (?w IS-PRED-OCCURRENCE
     :predicate        EXIST
                       :SUBJ    john_
                       :location of SUBJ     hospital_
  _  __(1-july-1997, 31-august-1997))
```

Fig. 3. Predicative occurrences and search patterns.

Search patterns' — i.e., the formal, NKRL counterparts of natural language queries — are now data structures that correspond to partially instantiated templates and that supply the general framework of information to be searched for, by filtering or unification, within an NKRL knowledge base. An example of search pattern, translating the query: "Was John at the hospital in July/August 1997?" — see the upper part of Figure 3 — is represented in the lower part of this last figure. The two timestamps associated with the pattern constitute now the 'search interval' that is used to limit the search for unification to the slice of time that it is considered appropriate to explore. In our example, the search pattern successfully unifies occurrence c2: in the absence of explicit, negative evidence, a situation is assumed to persist within the immediate temporal environment of the originating event, see [2].

In the CONCERTO version of NKRL, a Java module called FUM module (Filtering Unification Module) deals with search patterns. Unification is executed taking into account, amongst other things, the fact that a 'generic concept' included in the search pattern can unify one of its 'specific concepts' — or the instances (individuals) of a specific concept — included in a corresponding position of the occurrence. 'Generic' and 'specific' refer, obviously, to the structure of the NKRL concept ontology, i.e., H_CLASS. The inference level supplied by FUM is, however, only a first step towards more complex reasoning strategies. Some details about the high-level inference rules of NKRL can be found, e.g., in [7].

6. Conclusion, and Some Remarks and Comparisons

In a 'traditional' ontology, see [3], concepts are defined as frames according to two basic principles. The first is a *hierarchical* one, and it is materialised by the IsA link: it relates the concept to be defined to *all* the other concepts of the ontology trough the 'generic' (or 'subsumes'), 'specific' (or 'is-subsumed') and 'disjoint' relationships. The second is a *relational* principle and, via the 'attribute (property)-value' mechanism, relates the concept to be defined to *some* of the other concepts.

It is now evident that an organisation in terms of frames (or an equivalent one) is largely sufficient to provide a *static* definition of the concepts — i.e., a definition *a priori* of each concept considered in itself. We can, on the contrary, wonder if this sort of organisation can be sufficient to define the *dynamic behaviour* of the concepts, i.e., to describe the mutual relationships affecting *a posteriori* the concepts and their instances when they take part in some concrete action, situation etc. ('events'). If we want to represent a narrative fragment like "NMTV (an European media company) ... will develop a lap top computer system...", asserting that nmtv_ is an instance of the concept *company_* and that we must introduce an instance of a concept like *lap_top_pc* will not be sufficient. We must, in this case, have recourse to a most complex way of structuring the concepts that, as in NKRL, includes also a 'predicate' and the associated 'roles', the temporal co-ordinates, etc.

Of course, in the literature we find sometimes descriptions of frame-based systems trying to extend the attribute-value mechanism to produce some representations of 'events' according to an NKRL meaning. To code, in fact, some simple sell/purchase events, it is possible to add, in the frame for, e.g., *company_*, slots in the style of HasAcquired or AcquiredBy or, better, it is possible to define a new concept like *company_acquisition* with slots like NameOfTheCompany, Buyer, DateOfAcquisition, Price etc. In this way, the instances of *company_acquisition* could be sufficient to describe in a complete way a sell/purchase event for a company.

The limits of this approach are however evident. Restraining the description of sell/purchase events to the sole relationships between the buyer, the seller and the 'object' exchanged is, normally, only a very rough approximation of the original event, and a lot of useful information is, in this way, lost. It is very likely, in fact, that the original information about a company's sale was something in the style of: "Company X has sold its subsidiary Y to Z because the profits of Y had has fallen dangerously these last years due to a lack of investments" or, returning to a previous example, "NMTV will develop a lap top computer system to put controlled circulation magazines out of business". We are here in the domain of those 'connectivity phenomena' (like causality, goal, indirect speech, co-ordination and subordination etc.) I have evoked briefly in Section 4 and that are taken into account by the NKRL second order structures.

It is now easy to imagine, on the contrary, the awkward proliferation of totally *ad-hoc* slots that, sticking to the attribute-value paradigm, it would be necessary to introduce in order to approximate the real connectivity phenomena in the above examples. Trying to reduce the description of *events* to the description of *concepts* is then nothing that a further manifestation of the 'uniqueness syndrome' well-known in the Artificial Intelligence milieus. In NKRL, we make use in an integrated way of several sorts of representational principles, and several years of successful experimentation with the most different narrative situations are there to testify that this seems not to be a totally unreasonable approach.

References

1. Zarri, G.P., NKRL, a Knowledge Representation Tool for Encoding the 'Meaning' of Complex Narrative Texts, Natural Language Engineering - Special Issue on Knowledge Representation for NL Processing in Implemented Systems, 3 (1997) 231-253.
2. Zarri, G.P., Representation of Temporal Knowledge in Events: The Formalism, and Its Potential for Legal Narratives, Information & Communications Technology Law - Special Issue on Models of Time, Action, and Situations, 7 (1998) 213-241.
3. Fridman Noy, N., and Hafner, C.D., The State of the Art in Ontology Design - A Survey and Comparative Review, AI Magazine, 18(1997/3) 53-74.
4. Sowa, J.F., Conceptual Structures : Information Processing in Mind and Machine. Addison-Wesley. Reading (MA), 1994.
5. Brickley, D., Guha, R.V., eds. (1999) Resource Description Framework (RDF) Schema Specification. W3C, 1999 (http://www.w3.org/TR/WD-rdf-schema/).
6. Lassila, O., Swick, R.R., eds. Resource Description Framework (RDF) Model and Syntax Specification. W3C, 1999 (http://www.w3.org/TR/REC-rdf-syntax/).
7. Zarri, G.P., and Azzam, S., Building up and Making Use of Corporate Knowledge Repositories. In: Knowledge Acquisition, Modeling and Management - Proceedings of EKAW'97. Springer-Verlag, Berlin, 1997.

Using Agents for Concurrent Querying of Web-Like Databases via a Hyper-Set-Theoretic Approach

Vladimir Sazonov

Department of Computer Science
University of Liverpool, Liverpool L69 7ZF, U.K.
phone: (+44) 0151 794 6792, fax: (+44) 0151 794 3715
Web: http://www.csc.liv.ac.uk/~sazonov e-mail: V.Sazonov@csc.liv.ac.uk

Abstract. The aim of this paper is to present a brief outline of a further step in the ongoing work concerning the hyper-set-theoretic approach to (unstructured) distributed Web-like databases. The novel idea in this approach consists in using dynamically created mobile agents (processes) for more efficient querying such databases by exploiting concurrently distributed computational resources, potentially over the whole Internet.

1 Introduction

Querying and searching the World-Wide Web (WWW) or, more generally, *unstructured* or *Web-like Databases (WDB)* is one of the most important contemporary information processing tasks. There are several search engines such as Alta Vista, Lycos, etc., but their search can hardly be characterised as "goal-directed" and always "up to date". Also it is desirable not only to be able to find a list of Web-pages of potential interest, but to ask more complex queries allowing, additionally, reorganisation of Web data, as required. That is, the answer to a query should constitute a number of possibly newly created hyper-linked pages (re) constructed from some pages existing somewhere in WDB — very much in the same way as in a relational database. A new relation/WDB-page(s) (the answer to a query) is the result of reconstructing existing ones in the database. In fact, our approach to WDBs is *a natural generalisation of the traditional relational approach* and differs essentially from the other known approaches to semi-structured databases, being actually *(hyper) set-theoretic* one.

The starting point for this work was the characterisation of PTIME computability in terms of recursiveness over finite structures obtained by the author [26] and independently by N. Immerman, A. Livchak, M. Vardi and Y. Gurevich [14, 16, 21, 36, 14]. It should be mentioned, of course, the seminal previous work of R. Fagin [11] on describing NPTIME in terms of existential second-order logic. Such results were mainly considered in a *framework of an abstract approach to query languages for relational databases.* The subsequent work of the author on Bounded Set Theory [27, 28, 30, 18, 17], is a natural continuation and generalisation of these results for the case of more flexible structures such as hereditarily-finite sets which are more suitable for providing mathematical foundations of *complex* or *nested databases.* Later this approach absorbed, in [29], the idea of non-well-founded set (or *hyper-set*) theory introduced by P. Aczel [3]. This made the approach potentially applicable to (possibly distributed) *semistructured* or *Web-like databases* [19, 20, 31] with allowing cycles in hyper-links. Using *distributed, dynamically created agents* for more efficient querying of such databases by exploiting concurrently distributed computational resources (potentially over the whole Internet) is a further step to be developed within this approach.

Note, that our work *is not* intended to working out a *general theory* of agents. They are rather *a tool for achieving efficiency* in the querying process. However, an intermediate task consists of developing a new (or adapting some previously known) *theory or calculus of agents* suitable to our approach. (Cf. references at the end of Section 3.) In this short paper we can only outline the main (essentially inseparable) ideas: *hyper-set approach to WDB* and to *querying WDB*, and *using agents* (in the context of this hyper-set approach). Also, in spite of any declared allusions to reality of WWW, which seems very useful and thought provoking, the present paper is highly abstract (and simultaneously informal and sketchy by omitting the most of technicalities). But the author believes that the level of abstraction chosen is quite reasonable. The goal is not WWW itself, but some very general ideas around set-theoretic approach to WDB.

2 An Outline of Hyper-Set Approach to WDB

The popular idea of a *semi- (or un-) structured data base* which, unlike a relational database, has no rigid schema, has arisen in the database community rather recently (cf. e.g. [1, 2, 7, 22, 8]), particularly in connection with the Internet. These databases can also be naturally described as *Web-like* databases (WDB) [19, 20, 31–33]. The best example of such a database is the World-Wide Web itself understood as a collection of Web-pages (`html`-files) distributed by various *sites* over the world and arbitrarily *hyper-linked.*

We adopt here a deliberately simplified, very abstract picture which, however, contains the *main feature* of WWW as based on hyper-links between URLs of the Web-pages distributed potentially over the world. Having in mind set-theoretical approach (arisen quite independently of WWW in [27]) and contemporary ideas on semistructured databases (cf. op. cit.), we consider that *the visible part* of a WDB-page with the URL u is a (multi)set $\{\!|l_1, l_2, \dots, l_k|\!\}$ of words l_i (rather than a *sequence or list* $\langle l_1, l_2, \dots, l_k\rangle$) where $u \xrightarrow{l_i} v_i$ represent all the outgoing hyper-links from u to v_i. The URLs v_i are considered as *non-visible* part of the *page* $\{l_1 : v_1, l_2 : v_2, \dots, l_k : v_k\}$ (a set of pairs of the kind *label* : *URL* — an abstraction from an `html` file). Both l_i and corresponding v_i are present in this page having the URL u, but in the *main window* of a browser we do not see the URLs v_i; u being shown *outside* the window. Their role is very important for organisation of the information in WDB via *addressing* but quite *different* from that of the words l_i, the latter carrying the *proper information.* All words l_i on a Web-page are considered as (names or labels of) hyper-links, possibly trivial ones (e.g. links to an empty or non-existing page). By analogy with WWW we could underline $\underline{l_i}$ if $u \xrightarrow{l_i} v_i$ and v_i is "non-empty", i.e. there exists at least one hyper-link $v_i \xrightarrow{m} w$ outgoing from v_i (and it therefore makes sense to "click" on $\underline{l_i}$ to see all these m).

Such Web-like databases are evidently represented as directed *graphs with labelled edges* (or *labelled transition systems*), the labels l serving as the names of hyper-links or *references* $u \xrightarrow{l} v$ between graph vertices (URLs) u and v. As in any database, a query language and corresponding software querying system are needed to properly query and retrieve the required information from a WDB.

Let us assume additionally that each graph vertex (URL) u refers both to a *site* Site(u) ∈ SITES where the corresponding WDB-page is saved as an (`html`) file and to this WDB-page itself. Of course, different URLs may have the same site: Site(u_1) = Site(u_2). However it is not the most interesting case for us, it is quite possible in principle when there exists only one site for the whole WDB.

In the *hyper-set-theoretic* approach adopted here each graph vertex u (URL [19, 31] called also *object identity, Oid,* [1, 2, 7]) is considered as carrying some *information content* $(\!|u|\!)$ — the result of some abstraction from the concrete form of the graph. In principle, two Web-pages with different URLs u_1 and u_2 may have the same (*hereditarily,* under browsing starting from the given URLs u_1 and u_2) visible contents. This is written as $(\!|u_1|\!) = (\!|u_2|\!)$ or, equivalently, as $u_1 \sim u_2$ where $\sim$ is a *bisimulation equivalence relation* on graph vertices (which can be defined in graph-theoretic terms [3] independently of any hyper-set theory). Somewhat analogous approach based on a bisimulation relation is adopted in [7], but the **main idea of our approach** consists not only in respecting the bisimulation (or, actually, *informational equivalence*) relation, but in "considering" graph vertices *as abstract sets* $(\!|u|\!)$ within hyper-set theory [3]:

$$(\!|u|\!) = \{l : (\!|v|\!) \mid \text{WDB has a hyper-link } u \xrightarrow{l} v\}. \tag{1}$$

Thus, $(\!|u|\!)$ is a (hyper-) set of labelled elements $l : (\!|v|\!)$ (also hyper-sets, etc.) such that WDB has a hyper-link $u \xrightarrow{l} v$. This (actually uniquely satisfying (1)) set-theoretic *denotational semantics* of graph vertices has evidently a complete agreement with the meaning of the equality $(\!|u_1|\!) = (\!|u_2|\!)$ or bisimulation equivalence $u_1 \sim u_2$ briefly mentioned above. (Cf. details in [29, 19, 20, 31].)

Unlike the ordinary (Zermelo-Frenkel) set theory, cycles in the membership relation are here allowed. Hence, a simplest such "cycling" set is $\Omega = \{\Omega\}$ consisting exactly of itself. Such sets must be allowed because hyper-links (in a WDB or in WWW) may in general comprise arbitrary cycles. We have *crucial theoretical benefits* from this approach by using well understood languages (like Δ briefly discussed below) and ideas. Hyper-sets immediately arising from graph vertices via (1) are quite natural and pose no essential conceptual difficulty.

Returning to the graph view, *querying* of a WDB may be represented as consisting of the following steps:

- starting on a local site s (where the query $q(u)$ is evaluated) from an *initial* or *input URL u of a page* (saved as an `html` file on any remote site Site(u));
- *browsing* page-by-page via hyper-links (according to the query q);
- *searching* in pages arising in this process (according to the query q);

- *composing new (auxiliary) hyper-linked pages with their URLs* (on the basis of the previous steps) with one of these pages declared as *main answer* or *resulting (output or "title") page* for the query;
- all of this may result in *reorganising of the data* (at least locally on s by auxiliary pages located and hyper-linked between themselves and externally).

Of course, the user could habitually do all this job by hands. However, it is more reasonable to have corresponding implemented *query language and system* which would be able to formally express and evaluate any query q. This essentially differs from the ordinary search engines such as Alta Vista or Lycos, etc. Here we can use the advantages of our set theoretic approach and of the corresponding language Δ [28, 30, 19, 20], at least theoretically.

Again, the **key point of our approach** consists in co-existing and inter-playing *two* natural and related *viewpoints* for such a query q: graph- and set-theoretic ones. The above described steps result in some transformation (essentially a local extension) of the original WDB (WWW) graph. Usually, in semistructured databases *arbitrary* graph transformations are allowed [1, 2]. But if this process *respects the information contents* $\llparenthesis u \rrparenthesis$ of the WDB vertices (URLs) u then it should be restricted to be *bisimulation invariant* (as it was done also independently in [7], but without a sufficient stress on set theory) and therefore it could be also considered as a *set theoretic operation* q, i.e.

$$q : \llparenthesis \text{input URL} \rrparenthesis \longmapsto \llparenthesis \text{output URL} \rrparenthesis = q\llparenthesis \text{input URL} \rrparenthesis .$$

Originally, this approach arose exactly as a set-theoretical view [27, 28] with (at that time *acyclic*) graphs representing the ordinary or *well-founded hereditarily-finite sets* whose universe is denoted as **HF** [6]. Any database state may be considered as a set of data each element of which is also a further set of data, etc. (with always *finite depth of nesting* which is equal 4 in the case of a "flat" *relational database*). A generalised universe of *anti-founded hereditarily-finite (hyper) sets* is called **HFA**, and any query is considered as a map (set-theoretic operation — a well understood concept) $q : \mathbf{HF} \to \mathbf{HF}$ or $q : \mathbf{HFA} \to \mathbf{HFA}$.

A (version of) purely set-theoretic *query language* Δ allowing expression of queries q to a WDB has been developed (with a natural and very simple syntax whose main feature is the use of *bounded* quantifiers $\forall x \in t$ and $\exists x \in t$ and some other, essentially bounded, constructs; cf. e.g. [29, 30, 20, 19] for the details which will be also presented in the full version of the present paper). This language has *two kinds of semantics* in terms of: (i) *set-theoretic operations* q over **HFA** such us set union ("concatenation" of WDB-pages), etc., (*a high level* semantics — *for humans*) and (ii) corresponding *graph (WDB) transformers* Q (*a lower level* semantics — *for program implementation*) with a commutative diagram

$$\begin{tikzcd} \mathbf{HF} \arrow[r, "q"] & \mathbf{HF} \\ \mathcal{WDB} \arrow[u, "\llparenthesis - \rrparenthesis"] \arrow[r, "Q"] & \mathcal{WDB} \arrow[u, "\llparenthesis - \rrparenthesis"'] \end{tikzcd} \qquad q\llparenthesis \text{WDB} \rrparenthesis = \llparenthesis Q(\text{WDB}) \rrparenthesis \tag{2}$$

witnessing that both semantics agree (and Q is *bisimulation invariant* or *respecting informational equivalence*). Here $\mathcal{WDB}$ is the class of all WDBs (i.e. finite labelled graphs) with a distinguished URL (vertex) u in each, to which $\llparenthesis - \rrparenthesis$ is actually applied.

The *expressive power* of this language and its versions was completely characterised in terms of PRIME- (both for **HF** and **HFA**) and (N/D)LOGSPACE- (for **HF**) computability over graphs via (2) [27, 28, 19, 20, 18, 17]. It is because of flexibility, naturalness and reasonable level of abstractness of our set-theoretic approach to WDB (which seemingly nobody else used in the full power) such expressibility results where possible. Here it is probably suitable to mention, for the contrast, a quotation from [8], p. 14: "It should be noted that basic questions of expressive power for semistructured database query languages are still open".

There also exists a preliminary *experimental implementation* of Δ [32] developed (when the author worked) in the Program Systems Institute of Russian Academy of Sciences (in Pereslavl-Zalessky) as a query language for the "real" WWW which is based on the steps of browsing, searching, etc. described above.

3 Using Agents for Concurrent Querying WDB

What is new in the present approach. The *main problem* with implementing the set-theoretic language Δ is that (potentially) a *significant amount of browsing* may be required during query evaluation. In the real Internet the result of each mouse click on a hyper-link (downloading a page from remote site) is delayed by several seconds at best and the whole time of querying may take hours (despite all other pure computation steps being much faster). Therefore, some *radical innovation in the implementation is necessary* to overcome the problem of multiple browsing via the Internet (in a non-efficient sequential manner).

A well-known approach, used by most search engines such as Alta Vista, is that of creating (and permanently renewing) an *index file* of the WWW to which queries are addressed and some parts of which may be actually rather old.

The approach described here is aimed at "goal-directed" querying the *real* Web, *as it is* at the current moment, where *reorganising the data* (not only searching) is an additional and crucial feature of the query language Δ.

As a reasonable solution, we suggest using **Dynamic Agent Creation.** The set-theoretic language Δ appears very appropriate for this goal.

Agents as Δ-terms. Each agent, i.e. essentially a Δ-term (= a query $q = q(u)$ written in Δ-language) having an additional feature of an *active behaviour*, when starting querying from "his" local site s, may also *send* (via the Internet) to remote sites $s_1, s_2, \ldots$ some appropriate sub-terms $p_1, p_2, \ldots$ which, as *agents*, will work analogously on these remote sites, i.e., if necessary, will send new *descendant agents*, etc. Eventually, the main agent will collect all the data obtained from this process. Potentially, the *whole Internet* would *concurrently participate* in the *distributive* evaluation of the given query. We expect that this will essentially *accelerate querying*. One of medium-term goals is to establish the truth of this expectation in a theoretical framework. Another goal is producing an experimental agent based implementation of the language Δ to check practically this expectation and to get more experience for further developing this approach.

A Typical Example. To calculate the set-theoretic Δ-term

$$q = \{p(v) \mid v \in a\},$$

i.e. the "set of all $p(v)$ such that v is in a", this term, as an "agent", *sends many agents/sub-terms* $p(v)$ for all (URLs) v contained on the page a to the various, probably *remote*, sites Site(v) of the WWW to which all these (URLs) v refer. When each agent $p(v)$ finishes "his" computation (possibly with the help of their descendant agents), "he" will send the result to the "main agent" q, "who" will appropriately collect all the results together as a new (URL — the value of q — and corresponding `html` file of a) Web-page containing all the computed URLs $p(v)$ for all $v \in a$. Each agent q, $p(v)$, for all $v \in a$, etc. will resolve "his" own task, which may be not so difficult and burdensome for the site resources where the agent works, since "his" descendant agents will do the rest of the job possibly on other sites. If, by some reason, the "permission is denied" to send a descendant agent to a remote site, the parent agent will also take corresponding part of the job and will do it from "his" local site — by browsing, searching, etc.

We omit quite analogous and natural consideration of the other constructs of Δ which, in a sense, is almost "self suggesting" language for its parallel implementation by means of agents (especially in the case of a distributed WDB).

The further and most crucial goal of the described work therefore consists in *formalising the ideas above.* It may be done in the form of a specially elaborated (or suitably adapted for the query language Δ considered here) *calculus of agents* which describes as descendant agents are creating, moving around, doing their job, communicating their result to the "senior" agents by which they were created, until the result of the initial query is obtained. In particular, this will give *agent based graph transformer operational semantics* for the language Δ. Appropriate more detailed description of the corresponding agent calculus (which requires more place) will be presented in a full version of this paper.

Correctness of this semantics with respect to natural (hyper) set-theoretical semantics must be proved. It should be shown rigorously that the *time complexity* of this approach with agents is really better than that without agents (i.e. essentially with only one main agent which creates no sub-agents and makes all "his" job of browsing, searching and creating pages alone). In particular, it is necessary to formulate and develop a reasonable approach to time and space complexity for agent based querying (cf. the next paragraph), to estimate and compare time and space complexity of queries either when agents are used or not. In the ideal, the resulting agent calculus should be *capable of being implemented* as a working *query system* to a WDB, say, to the WWW.

Note that an appropriate abstraction from querying of the WWW should be found. For example, in reality, each step of browsing requires some physical, actually unpredictable time. For simplicity it may be taken that each step of browsing or exchanging information between agents takes *one unit of time* and all other computation steps (of searching some information trough Web-pages and composing new ones) cost nothing (in comparison with the browsing). Also when several agents are sent to the same Web-site for their work their activity may be considered either as sequential or in an interleaving manner according to communication with other agents. Space complexity also may be considered in a simplified version, e.g. as the maximum number of agents working simultaneously. The simultaneity concept should be also defined (approximated) appropriately as the run of time in the model may not correspond to the real one, as we noted above. On the other hand, how much of memory resource each agent needs on its site (server) could additionally be taken into account.

Some related work (i) on *communication and concurrency* by R. Milner and P. Aczel, [23, 4, 5], (ii) on *mobile agents (ambients)* by R. Milner et al. [24], L. Cardelli [8], C. Fournet et al. [13], (iii) on *logical specification* of agents' behaviour by M. Fisher and M. Wooldridge [12], (iv) on *distributive querying* by D. Suciu [34, 35] and A. Sahuguet et al. [25], (v) on corresponding application of Y. Gurevich's *Abstract State Machines* [10] and (vi) on a project *LOGIC Programming for the World-Wide WEB* by A. Davison et al. [9] could be useful in our specific set-theoretic context either from the ideological point of view, by direct using corresponding formalisms, or by developing some analogues.

Further perspectives from the practical point of view. It is clear, that *the whole approach depends on giving permissions* by various Web sites for agents to work there. For a WDB on a local net or on an *Intra*net this problem can be resolved by an agreement with the administrator of the net. However, for the case of the global Internet the whole Internet community should agree on a *general standard* for such permissions with an appropriate guarantee of no danger from the agents for various sites over the World. Of course this may depend on whether this approach will be sufficiently successful for local nets, as well as on the current state of affairs concerning programming in the Internet.

4 Conclusion

The starting point for this research was a mostly theoretical one related to descriptive complexity theory and to extending its methods to relational, nested, complex and Web-like distributed databases grounded on (hyper) set theory. Our present work is also theoretical, but it is directed towards the problem of developing a necessary radical step for more efficient implementation. Considering dynamically created agents in a full generality, working and communicating concurrently and distributively over the Internet is not the immediate goal of research here, but rather represents the machinery for achieving the efficiency of querying. As a result this work may be of direct benefit to research communities on semi-structured and distributed databases and on multi-agent systems. It also has the potential of longer term benefit to the Internet community. More concretely, it may lead to better understanding of the basic principles of efficient querying of WDB and to an experimental prototype of such a query system.

5 Acknowledgements

The author is very grateful to Michael Fisher, Alexander Bolotov and Chris Brand for fruitful discussions on the approach presented and for the help in polishing English. He is especially thankful to his colleagues from Pereslavl-Zalessky (Russia) Alexander Leontjev, Alexei Lisitsa and Yuri Serdyuk actively participating in set-theoretic approach to WDB.

References

1. S. Abiteboul, Querying semi-structured data. *Database Theory — ICDT'97, 6th International Conference*, Springer, 1997, 1–18
2. S. Abiteboul and V. Vianu, Queries and computation on the Web. *Database Theory — ICDT'97, 6th International Conference*, Springer, 1997
3. P. Aczel, *Non-Well-Founded Sets*, CSLI Lecture Notes. No **14** (1988)
4. P. Aczel, Final Universes of Processes, *9th Internat. Conf. on Math. Foundations of Programming Semantics*, ed. S.Brookes, et al. Springer LNCS **802** (1994), 1-28

5. P. Aczel, Lectures on Semantics: The initial algebra and final coalgebra perspectives, *Logic of Computation*, ed. H. Schwichtenberg, Springer (1997)
6. J.K. Barwise, *Admissible Sets and Structures.* Springer, Berlín, 1975
7. P. Buneman, S. Davidson, G. Hillebrand, D. Suciu, A query Language and Optimisation Techniques for Unstructured Data. *Proc. of SIGMOD*, San Diego, 1996
8. L. Cardelli, Semistructured Computation, Febr. 1, 2000; and other papers on mobile ambient calculi available via http://www.luca.demon.co.uk/
9. A. Davison, L. Sterling and S.W. Loke, *LogicWeb: LOGIC Programming for the World-Wide WEB*, http://www.cs.mu.oz.au/~swloke/logicweb.html
10. R. Eschbach, U. Glässer, R. Gotzhein, and A. Prinz, On the Formal Semantics of Design Languages: A compilation approach using Abstract State Machines, Springer LNCS **1912** 2000, 131–151,
11. R. Fagin, Generalized first order spectra and polynomial time recognizable sets, *Complexity of Computations*, SIAM — AMS Proc. **7**, 1974, 43–73
12. M. Fisher and M. Wooldridge, On the Formal Specification and Verification of Multi-Agent Systems. *Intern. J. of Cooperative Information Systems*, 6(1), 1997
13. C. Fournet, G. Gonthier, J-J. Lévy, L. Maranget, and D. Rémy, A Calculus of Mobile Agents, *CONCUR'96*, Springer LNCS **1119** (1996), 406–421
14. Y. Gurevich, Algebras of feasible functions. *FOCS* **24** (1983), 210–214
15. Y. Gurevich at al., Web-site on ASM: http://www.eecs.umich.edu/gasm/
16. N. Immerman, Relational queries computable in polynomial time, *Proc. 14th. ACM Symp. on Theory of Computing*, (1982) 147–152
17. A. Leontjev and V. Sazonov, Capturing LOGSPACE over Hereditarily-Finite Sets, *FoIKS'2000*, Springer LNCS **1762** (2000), 156–175
18. A. Lisitsa and V. Sazonov, Delta-languages for sets and LOGSPACE computable graph transformers. *Theoretical Computer Science* **175**, 1 (1997), 183–222
19. A. Lisitsa, and V. Sazonov, Bounded Hyper-set Theory and Web-like Data Bases, *KGC'97*, Springer LNCS **1289** (1997), 172–185
20. A. Lisitsa and V. Sazonov, Linear ordering on graphs, anti-founded sets and polynomial time computability, *Theoretical Computer Science* **224**, 1–2 (1999) 173-213
21. A.B. Livchak, Languages of polynomial queries. *Raschet i optimizacija teplotehnicheskih ob'ektov s pomosh'ju EVM*, Sverdlovsk, 1982, p. 41 (in Russian)
22. A.O. Mendelzon, G.A. Mihaila, T. Milo, Querying the World Wide Web, Int. J. on Digital Libraries 1(1): 54–67 (1997)
23. R. Milner, *Communication and Concurrency*, Prentice Hall, 1989
24. R. Milner, RJ. Parrow and D. Walker, A calculus of mobile processes, Parts 1–2, *Information and Computation*, 100(1), 1–66, 1992 (and other papers on Action calculi and the pi-calculus available via http://www.cl.cam.ac.uk/users/rm135/)
25. A. Sahuguet, B. Pierce and Val Tannen, Distributed Query Optimization: Can Mobile Agents Help? 2000, cf. http://db.cis.upenn.edu/Publications/
26. V. Sazonov, Polynomial computability and recursivity in finite domains. *Elektronische Informationsverarbeitung und Kybernetik*, **16** (7) (1980) 319–323
27. V. Sazonov, Bounded set theory and polynomial computability. *All Union Conf. Appl. Logic., Proc.* Novosibirsk, 1985, 188–191 (In Russian) Cf. also essentially extended English version with new results in *FCT'87*, LNCS **278** (1987), 391–397
28. V. Sazonov, Hereditarily-finite sets, data bases and polynomial-time computability. *TCS*, **119** Elsevier (1993), 187–214
29. V. Sazonov, A bounded set theory with anti-foundation axiom and inductive definability, *CSL'94* Springer LNCS **933** (1995) 527–541
30. V. Sazonov, On Bounded Set Theory. Invited talk on the *10th International Congress on Logic, Methodology and Philosophy of Sciences, in Volume I: Logic and Scientific Method*, Kluwer Academic Publishers, 1997, 85–103
31. V. Sazonov, Web-like Databases, Anti-Founded Sets and Inductive Definability, *Programmirovanie*, 1999, N5, 26-43 (In Russian; cf. also English version of this J.).
32. V. Sazonov and Yu. Serdyuk, Experimental implementation of set-theoretic query language Delta to Web-like databases (in Russian), *Programmnye Systemy, Teoreticheskie Osnovy i prilozhenija,* RAN, Institut Programmnyh System, Moskva, Nauka, Fizmatlit, 1999
33. Yu. Serdyuk, Partial Evaluation in a Set-Theoretic Query Language for WWW, *FoIKS'2000*, Springer LNCS **1762** (2000), 260–274
34. D. Suciu, Distributed Query Evaluation on Semistructured Data, 1997. Available from http://www.research.att.com/~suciu
35. D. Suciu, Query decomposition and view maintenance for query languages for unstructured data, *Proc. of the Internat. Conf. on Very Large Data Bases*, 1996
36. M.Y. Vardi, The complexity of relational query languages. *Proc. of the 14th. ACM Symp. on Theory of Computing*, (1982) 137–146

Reexecution-Based Analysis of Logic Programs with Delay Declarations*

Agostino Cortesi[1], Sabina Rossi[1], and Baudouin Le Charlier[2]

[1] Dipartimento di Informatica, Università Ca' Foscari di Venezia
via Torino 155, 30172 Venezia, Italy
fax: +39 041 2908419
{cortesi,srossi}@dsi.unive.it
[2] Institut d'Informatique, University of Namur
21 rue Grandgagnage, B-5000 Namur, Belgium
e-mail: ble@info.fundp.ac.be

Abstract. A general semantics-based framework for the analysis of logic programs with delay declarations is presented. The framework incorporates well known refinement techniques based on reexecution. The concrete and abstract semantics express both deadlock information and qualified answers.

1 Introduction

In order to get more efficiency, users of current logic programming environments, like Sictus-Prolog [13], Prolog-III, CHIP, SEPIA, etc., are not forced to use the classical Prolog left-to-right scheduling rule. Dynamic scheduling can be applied instead where atom calls are delayed until their arguments are sufficiently instantiated, and procedures are augmented with delay declarations. The analysis of logic programs with dynamic scheduling was first investigated by Marriott *et al.* in [18, 11]. A more general (denotational) semantics of this class of programs, extended to the general case of CLP, has been presented by Falaschi *et al.* in [12], while verification and termination issues have been investigated by Apt and Luitjes in [2] and by Marchiori and Teusink in [17], respectively.

In this paper we discuss an alternative, strictly operational, approach to the definition of concrete and abstract semantics for logic programs with delay declarations. The approach uses the reexecution technique which exploits the well known property of logic programming that a goal may be reexecuted arbitrarily often without affecting the semantics of the program. This property has been pointed out since 1987 by Bruynooghe [3, 4] and subsequently used in abstract interpretation to improve the precision of the analysis [15].

The main intuitions behind our proposal can be summarized as follows:

- to define in a uniform way concrete, collecting, and abstract semantics, in the spirit of [14]: this allows us to easily derive correctness proofs of the whole analyses;
- to define the analysis as an extension of the framework depicted in [14]: this allows us to reuse existing code for program analysis, with minimal additional effort;
- to explicitly derive deadlock information (possible deadlock and deadlock freeness) producing, as a result of the analysis, an approximation of concrete qualified answers;
- to apply the reexecution technique developed in [15], that plays a crucial role here: if during the execution of an atom a a deadlock occurs, then a is allowed to be reexecuted at a subsequent step.

* Partially supported by Italian MURST Projects "Interpretazione Astratta, Type Systems e Analisi Control-Flow", and "Certificazione automatica di programmi mediante interpretazione astratta".

The main difference between our approach and the ones already presented in the literature is that we are mainly focussed on analysis issues, in particular on deadlock freeness analysis. This motivates the choice of a strictly operational approach, where deadlock information is explicitly maintained.

This paper illustrates the crucial steps toward the definition and implementation of an extension of the GAIA abstract interpreter [14] to deal with dynamic scheduling. It mainly focuses on the (concrete and abstract) semantics upon which a generic fixpoint algorithm is defined.

The main idea is partitioning literals of a goal g into three sets: literals which are delayed, literals which are not delayed and have not been executed yet, and literals which are allowed to be reexecuted as they are not delayed but have already been executed before and fallen into deadlock[1].

The rest of the paper is organized as follows. In the next section we recall some basic notions about logic programs with delay declarations. Section 3 depicts the concrete operational semantics which serves as a basis for the new abstract semantics introduced in Section 4. Correctness of our generic fixpoint algorithm is discussed. Section 5 concludes the paper.

2 Logic Programs with Delay Declarations

Logic programs with delay declarations consist of two parts: a logic program and a set of delay declarations, one for each of its predicate symbols.

A *delay declaration* associated for an n-ary predicate symbol p has the form

$$\texttt{DELAY}\ \ p(x_1,\ldots,x_n)\ \ \texttt{UNTIL}\ \ Cond(x_1,\ldots,x_n)$$

where $Cond(x_1,\ldots,x_n)$ is a formula in some assertion language. We are not concerned here with the syntax of this language since it is irrelevant for our purposes. The meaning of such a delay declaration is that an atom $p(t_1,\ldots,t_n)$ can be selected in a query only if the condition $Cond(t_1,\ldots,t_n)$ is satisfied. In this case we say that the atom $p(t_1,\ldots,t_n)$ *satisfies* its delay declaration.

A derivation of a program augmented with delay declarations *succeeds* if it ends with the empty goal; while it *deadlocks* if it ends with a non-empty goal no atom of which satisfies its delay declaration. Both successful and deadlocked derivations compute *qualified answers*, i.e., pairs of the form $\langle\theta,d\rangle$ where d is the last goal (that is a possibly empty sequence of delayed atoms) and θ is the substitution obtained by concatenating the computed mgu's from the initial goal. Notice that, if $\langle\theta,d\rangle$ is a qualified answer for a successful derivation then d is the empty goal and θ restricted to the variables of the initial goal is the corresponding computed answer substitution. We denote by $qans_P(g)$ the set of qualified answers for a goal g and a program P.

We restrict our attention to delay declarations which are *closed under instantiation*, i.e., if an atom satisfies its delay declaration then also all its instances do. Notice that this is the choice of most of the logic programming systems dealing with delay declarations such as IC-Prolog, NU-Prolog, Prolog-II, Sicstus-Prolog, Prolog-III, CHIP, Prolog M, SEPIA, etc.

The following example illustrates the use of delay declarations in logic programming.

Example 1. Consider the program PERMUTE discussed by Naish in [19].

```
% perm(Xs,Ys) ← Ys is a permutation of the list Xs
  perm(Xs,Ys) ← Xs = [ ], Ys = [ ].
  perm(Xs,Ys) ← Xs = [X|X1s], delete(X,Ys,Zs), perm(X1s,Zs).

% delete(X,Ys,Zs) ← Zs is the list obtained by removing X from the list Ys
  delete(X,Ys,Zs) ← Ys = [X|Zs].
  delete(X,Ys,Zs) ← Ys = [X1|Y1s], Zs = [X1|Z1s], delete(X,Y1s,Z1s).
```

Clearly, the relation declaratively given by perm is symmetric. Unfortunately, the behavior of the program with Prolog (using the leftmost selection rule) is not. In fact, given the query

$$Q_1 := \leftarrow \texttt{perm(Xs,[a,b])}.$$

Prolog will correctly backtrack through the answers Xs = [a, b] and Xs = [b, a]. However, for the query

$$Q_2 := \leftarrow \texttt{perm([a,b],Xs)}.$$

[1] This partitioning dramatically simplifies both concrete and abstract semantics with respect to the approach depicted in [8], where a very preliminary version of this work was presented.

$P \in$ *Programs*	$P ::= pr_1, \ldots, pr_n\ (n > 0)$
$pr \in$ *Procedures*	$pr ::= c_1, \ldots, c_n\ (n > 0)$
$c \in$ *Clauses*	$c ::= h :- g.$
$h \in$ *ClauseHeads*	$h ::= p(x_1, \ldots, x_n)\ (n \geq 0)$
$g \in$ *LiteralSequences*	$g ::= l_1, \ldots, l_n\ (n \geq 0)$
$l \in$ *Literals*	$l ::= a \mid b$
$a \in$ *Atoms*	$a ::= p(x_{i_1}, \ldots, x_{i_n})\ (n \geq 0)$
$b \in$ *Built-ins*	$b ::= x_i = x_j \mid x_{i_1} = f(x_{i_2}, \ldots, x_{i_n})$
$p \in$ *ProcedureNames*	
$f \in$ *Functors*	
$x_i \in$ *ProgramVariables*	

Fig. 1. Abstract Syntax of Normalized Programs

Prolog will first return the answer Xs = [a, b] and on subsequent backtracking will fall into an infinite derivation without returning answers anymore.

For languages with delay declarations the program PERMUTE behaves symmetrically. In particular, if we consider the delay declarations:

```
DELAY perm(Xs,_) UNTIL nonvar(Xs).
DELAY delete(_,_,Zs) UNTIL nonvar(Zs).
```

the query Q_2 above does not fall into a deadlock. ■

Under the assumption that delay declarations are closed under instantiation, the following result, which is a variant of Theorem 4 in Yelick and Zachary [21], holds.

Theorem 1. *Let P be a program augmented with delay declarations, g be a goal and g' be a permutation of g. Then $qans_P(g)$ and $qans_P(g')$ are equals modulo the ordering of delayed atoms.*

It follows that both successful and deadlocked derivations are "independent" from the choice of the selection rule. Moreover, Theorem 1 allows us to treat goals as multisets instead of sequences of atoms.

3 The Concrete Operational Semantics

In this section we describe a concrete operational semantics for pure Prolog augmented with delay declarations. The concrete semantics is the link between the standard semantics of the language and the abstract one. We assume a preliminary knowledge of logic programming (see, [1, 16]).

Programs Programs are assumed to be normalized according to the syntax given in Fig. 1. The variables occurring in a literal are distinct; distinct procedures have distinct names; all clauses of a procedure have exactly the same head; if a clause uses m different program variables, these variables are $x_1, \ldots, x_m$. If $g := a_1, \ldots, a_n$ we denote by $g \setminus a_i$ the goal $g' := a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n$.

Program Substitutions We assume the existence of two disjoint and infinite sets of variables: *program variables*, which are ordered and denoted by x_1, x_2, $\ldots$, x_i, $\ldots$, and *standard variables* which are denoted by letters y and z (possibly subscripted). Programs are built using program variables only.

A program substitution is a set $\{x_{i_1}/t_1, \ldots, x_{i_n}/t_n\}$, where $x_{i_1}, \ldots, x_{i_n}$ are distinct program variables and t_1, $\ldots$, t_n are terms (built with standard variables only). Program substitutions are not substitutions in the usual sense; they are best understood as a form of program store which expresses the state of the computation at a given program point. It is meaningless to compose them as usual substitutions. The domain of a program substitution $\theta = \{x_{i_1}/t_1, \ldots, x_{i_n}/t_n\}$, denoted by $dom(\theta)$, is the set of program variables $\{x_{i_1}, \ldots, x_{i_n}\}$. The application $x_i\theta$ of a program substitution θ to a program variable x_i is defined only if $x_i \in dom(\theta)$: it denotes the term bound to x_i in θ. Let D be a finite set of program variables. We denote by PS_D the set of program substitutions whose domain is D.

Concrete Behaviors The notion of concrete behavior provides a mathematical model for the input/output behavior of programs. To simplify the presentation, we do not parameterize the semantics with respect to programs. Instead, we assume given a fixed underlying program P augmented with delay declarations.

We define a *concrete behavior* as a relation from input states to output states as defined below. The *input states* have the form

- $\langle\theta, p\rangle$, where p is the name of a procedure and θ is a program substitution also called activation substitution. Moreover, $\theta \in PS_{\{x_1,\ldots,x_n\}}$, where $x_1,\ldots,x_n$ are the variables occurring in the head of every clause of p.

The *output states* have the form

- $\langle\theta', \kappa\rangle$, where $\theta' \in PS_{\{x_1,\ldots,x_n\}}$ and κ is a deadlock state, i.e., it is an element from the set $\{\delta, \nu\}$, where δ stands for *definite deadlock*, while ν stands for *no deadlock*. In case of no deadlock, θ' restricted to the variables $\{x_1,\ldots,x_n\}$ is a computed answer substitution (the one corresponding to a successful derivation), while in case of deadlock, θ' is the substitution part of a qualified answer to p and coincides with a partial answer substitution for it.

We use the relation symbol $\longmapsto$ to represent concrete behaviors, i.e., we write $\langle\theta, p\rangle \longmapsto \langle\theta', \kappa\rangle$: this notation emphasizes the similarities between this concrete semantics and the structural operational semantics for logic programs defined in [15]. Concrete behaviors are intended to model successful and deadlocked derivations of atomic queries.

Concrete Semantic Rules The concrete semantics of an underlying program P with delay declarations is the least fixpoint of a continuous transformation on the set of concrete behaviors. This transformation is defined in terms of semantic rules that naturally extend concrete behaviors in order to deal with clauses and goals. In particular, a concrete behavior is extended through intermediate states of the form $\langle\theta, c\rangle$ and $\langle\theta, g\_d, g\_e, g\_r\rangle$, where c is a clause and g_d, g_e, g_r is a partition of a goal g such that: g_d contains all literals in g which are delayed, g_e contains all literals in g which are not delayed and have not been executed yet, g_r contains all literals in g which are allowed to be reexecuted, i.e., all literals that are not delayed and have already been executed but fallen into a deadlock.

- Each pair $\langle\theta, c\rangle$, where c is a clause, $\theta \in PS_{\{x_1,\ldots,x_n\}}$ and $x_1,\ldots,x_n$ are the variables occurring in the head of c, is related to an output state $\langle\theta', \kappa\rangle$, where $\theta' \in PS_{\{x_1,\ldots,x_n\}}$ and $\kappa \in \{\delta, \nu\}$ is a deadlock state;
- Each tuple $\langle\theta, g\_d, g\_e, g\_r\rangle$, where $\theta \in PS_{\{x_1,\ldots,x_m\}}$ and $x_1,\ldots,x_m$ are the variables occurring in (g_d, g_e, g_r), is related to an output state $\langle\theta', \kappa\rangle$, where $\theta' \in PS_{\{x_1,\ldots,x_m\}}$ and $\kappa \in \{\delta, \nu\}$ is a deadlock state.

We briefly recall here the concrete operations which are used in the definition of the concrete semantic rules depicted in Fig. 2. The reader may refer to [14] for a complete description of all operations but the last one, SPLIT, that is brand new.

- EXTC is used at clause entry: it extends a substitution on the set of variables occurring in the body of the clause.
- RESTRC is used at clause exit: it restricts a substitution on the set of variables occurring in the head of the clause.
- RETRG is used when a literal l occurring in the body of a clause is analyzed. Let $\{x_{i_1},\ldots,x_{i_n}\}$ be the set of variables occurring in l. This operation expresses a substitution on variables $x_{i_1},\ldots,x_{i_n}$ in terms of the formal parameters $x_1,\ldots,x_n$.
- EXTG it is used to combine the analysis of a built-in or a procedure call (expressed in terms of the formal parameters $x_1,\ldots,x_n$) with the activating substitution.
- UNIF-FUNC and UNIF-VAR are the operations that actually perform the unification of equations of the form $x_i = x_j$ or $x_{i_1} = f(x_{i_2},\ldots,x_{i_n})$, respectively.
- SPLIT is a new operation: given a substitution θ and a goal g, it partitions g into the set of atoms g_d which do not satisfy the corresponding delay declarations, and then are not executable, and the set of atoms g_e which satisfy the corresponding delay declarations, and then are executable.

The definition of the concrete semantic rules proceeds by induction on the syntactic structure of program P. Rule $\mathbf{R}_1$ defines the result of executing a procedure call: this is obtained by executing any clause defining it. Rule $\mathbf{R}_2$ defines the result of executing a clause: this is obtained by executing its body under the same input substitution after splitting the body into two parts: executable literals and delayed literals. Rule $\mathbf{R}_3$ defines

$$\textbf{R1}\ \frac{\begin{array}{c}c \text{ is a clause defining } p\\ \langle\theta, c\rangle \longmapsto \langle\theta', \kappa\rangle\end{array}}{\langle\theta, p\rangle \longmapsto \langle\theta', \kappa\rangle} \qquad \textbf{R2}\ \frac{\begin{array}{c}c := h : -g\\ \theta_1 = \mathtt{EXTC}(c, \theta)\\ \langle g\_d, g\_e\rangle = \mathtt{SPLIT}(\theta_1, g)\\ \langle\theta_1, g\_d, g\_e, <>\rangle \longmapsto \langle\theta_2, \kappa\rangle\\ \theta' = \mathtt{RESTRC}(c, \theta_2)\end{array}}{\langle\theta, c\rangle \longmapsto \langle\theta', \kappa\rangle}$$

$$\textbf{R3}\ \frac{}{\langle\theta, <>, <><>, \rangle \longmapsto \langle\theta, \nu\rangle} \qquad \textbf{R4}\ \frac{\text{either } g\_d \neq <> \text{ or } g\_r \neq <>}{\langle\theta, g\_d, <>, g\_r\rangle \longmapsto \langle\theta, \delta\rangle}$$

$$\textbf{R5}\ \frac{\begin{array}{c}\bar{g}\_e := g\_e \setminus b\\ b := x_i = x_j\\ \theta_1 = \mathtt{RESTRG}(b, \theta)\\ \theta_2 = \mathtt{UNIF\_VAR}(\theta_1)\\ \theta_3 = \mathtt{EXTG}(b, \theta, \theta_2)\\ \langle\theta_3, g\_r\rangle \longmapsto_r \langle\theta_4, \bar{g}\_r\rangle\\ \langle\bar{g}\_d, \bar{g}'\_e\rangle = \mathtt{SPLIT}(\theta_4, g\_d)\\ \langle\theta_4, \bar{g}\_d, \bar{g}\_e \cup \bar{g}'\_e, \bar{g}_r\rangle \longmapsto \langle\theta', \kappa\rangle\end{array}}{\langle\theta, g\_d, g\_e, g\_r\rangle \longmapsto \langle\theta', \kappa\rangle} \qquad \textbf{R6}\ \frac{\begin{array}{c}\bar{g}\_e := g\_e \setminus b\\ b := x_i = f(x_{i_1}, \ldots, x_{i_n})\\ \theta_1 = \mathtt{RESTRG}(b, \theta)\\ \theta_2 = \mathtt{UNIF\_FUNC}(b, \theta_1)\\ \theta_3 = \mathtt{EXTG}(b, \theta, \theta_2)\\ \langle\theta_3, g\_r\rangle \longmapsto_r \langle\theta_4, \bar{g}\_r\rangle\\ \langle\bar{g}\_d, \bar{g}'\_e\rangle = \mathtt{SPLIT}(\theta_4, g\_d)\\ \langle\theta_4, \bar{g}\_d, \bar{g}\_e \cup \bar{g}'\_e, \bar{g}_r\rangle \longmapsto \langle\theta', \kappa\rangle\end{array}}{\langle\theta, g\_d, g\_e, g\_r\rangle \longmapsto \langle\theta', \kappa\rangle}$$

$$\textbf{R7}\ \frac{\begin{array}{c}\bar{g}\_e := g\_e \setminus a\\ a := p(x_{i_1}, \ldots, x_{i_n})\\ \theta_1 = \mathtt{RESTRG}(a, \theta)\\ \langle\theta_1, p\rangle \longmapsto \langle\theta_2, \nu\rangle\\ \theta_3 = \mathtt{EXTG}(a, \theta, \theta_2)\\ \langle\theta_3, g\_r\rangle \longmapsto_r \langle\theta_4, \bar{g}\_r\rangle\\ \langle\bar{g}\_d, \bar{g}'\_e\rangle = \mathtt{SPLIT}(\theta_4, g\_d)\\ \langle\theta_4, \bar{g}\_d, \bar{g}\_e \cup \bar{g}'\_e, \bar{g}_r\rangle \longmapsto \langle\theta', \kappa\rangle\end{array}}{\langle\theta, g\_d, g\_e, g\_r\rangle \longmapsto \langle\theta', \kappa\rangle} \qquad \textbf{R8}\ \frac{\begin{array}{c}\bar{g}\_e := g\_e \setminus a\\ a := p(x_{i_1}, \ldots, x_{i_n})\\ \theta_1 = \mathtt{RESTRG}(a, \theta)\\ \langle\theta_1, p\rangle \longmapsto \langle\theta_2, \delta\rangle\\ \theta_3 = \mathtt{EXTG}(a, \theta, \theta_2)\\ \langle\theta_3, g\_r.a\rangle \longmapsto_r \langle\theta_4, \bar{g}\_r\rangle\\ \langle\bar{g}\_d, \bar{g}'\_e\rangle = \mathtt{SPLIT}(\theta_4, g\_d)\\ \langle\theta_4, \bar{g}\_d, \bar{g}\_e \cup \bar{g}'\_e, \bar{g}_r\rangle \longmapsto \langle\theta', \kappa\rangle\end{array}}{\langle\theta, g\_d, g\_e, g\_r\rangle \longmapsto \langle\theta', \kappa\rangle}$$

Fig. 2. Concrete Semantic Rules

the result of executing the empty goal, generating a successful output substitution. Rule $\mathbf{R}_4$ defines a deadlock situation that yields a definite deadlock information δ. Rules $\mathbf{R}_5$ to $\mathbf{R}_8$ specify the execution of a literal. First, the literal is executed producing an output substitution θ_3; then reexecutable atoms are (re)executed through the auxiliary relation $\langle\theta_3, g\_r\rangle \longmapsto_r \langle\theta_4, \bar{g}\_r\rangle$: its effect is to refine θ_3 into θ_4 and to remove from g_r the atoms that are completely solved in θ_4 returning the new list of reexecutable atoms $\bar{g}\_r$; finally, the sequence of delayed atoms with the new substitution θ_4 is partitioned in two sets: the atoms that are still delayed and those that have been awakened. Rules $\mathbf{R}_5$ and $\mathbf{R}_6$ specify the execution of built-ins and use the unification operations. Rules $\mathbf{R}_7$ and $\mathbf{R}_8$ define the execution of an atom a in the case that a has not been considered yet. The first rule applies when the execution of a is deadlock free; while the second rule applies when the execution of a with the current activation substitution falls into deadlock: in this case, a is moved in the reexecutable atoms list.

Because of lake of space, we do not specify here the reexecutable rules defining the auxiliary relation $\longmapsto_r$, which can be easily obtained following the methodology defined in [15].

The concrete semantics of a program P with delay declarations is defined as a fixpoint of this transition system. We can prove that this operational semantics is safe with respect to the standard resolution of programs with delay declarations.

4 Collecting and Abstract Semantics

As usual in the *Abstract Interpretation* approach [9,10], in order to define an abstract semantics we proceed in three steps. First, we depict a collecting semantics, by lifting the concrete semantics to deal with sets of substitutions. Then, any abstract semantics will be defined as an abstraction of the collecting semantics: it is sufficient to provide an abstract domain that enjoys a Galois connection with the concrete domain $\wp(Subst)$, and a suite of abstract operations that safely approximate the concrete ones. Finally, we draw an algorithm to compute a (post-)fixpoint of an abstract semantics defined this way.

The collecting semantics can be trivially obtained from the concrete one by

- replacing substitutions with sets of substitutions;
- using μ, standing for *possible deadlock*, instead of δ;
- redefining all operations in order to deal with sets of substitutions (as done in [14]).

In particular, the collecting version of operation SPLIT, given a set of substitutions Θ, will partition a goal g into the set of atoms g_d which do not satisfy the corresponding delay declarations for some $\theta \in \Theta$, and the set of atoms g_e which do satisfy the corresponding delay declarations for some $\theta \in \Theta$. Notice that this approach is sound, i.e., if an atom is executed at the concrete level then it will be also at the abstract level. However, since some atoms can be put both in g_d and in g_e some level of imprecision could arise.

Once the collecting semantics is fixed, deriving abstract semantics is almost an easy job. Any domain abstracting substitutions can be used to describe abstract activation states. Similarly to the concrete case, we distinguish among input states, e.g., $\langle \beta, p \rangle$ where β is an approximation of a set of activation substitutions, and output states, e.g., $\langle \beta', \kappa \rangle$ where β' is an approximation of a set of output substitutions and $\kappa \in \{\mu, \nu\}$ is an abstract deadlock state. Clearly, the accuracy of deadlock analysis will depend on the matching between delay declarations and the information represented by the abstract domains. It is easy to understand, by looking at the concrete semantics presented above, that very few additional operations should be implemented on an abstract substitution domain like the ones in [6, 7, 14], while a great amount of existing specification and coding can be reused for free.

Fig. 3 reports the final step in the Abstract Interpretation picture described above: an abstract transformation that abstracts the concrete semantics rules. The abstract semantics is defined as a post-fixpoint of transformation *TAB* on sets of abstract tuples, *sat*, as defined in the picture. An algorithm computing the abstract semantics can be defined by simple modification of the reexecution fixpoint algorithm presented in [15]. The reexecution function T_r is in the spirit of [15]. It uses the abstract operations REFINE and RENAME, where

- REFINE is used to refine the result β of executing an atom by combining it with the results obtained by reexecution of atoms in the reexecutable atom lists starting from β itself;
- RENAME is used after reexecution of an atom a: it expresses the result of reexecution in terms of the variables $x_{i_1}, \ldots, x_{i_n}$ occurring in a.

As already observed before, most of the operations that are used in the algorithm are simply inherited from the GAIA framework [14]. The only exception is SPLIT, which depends on a given set of delay declarations.

The correctness of the algorithm can be proven the same way as in [14] and [15]. What about termination? The execution of T_b terminates since the number of literals in g_d and g_e decreases of exactly one at each recursive call. The fact that the execution of T_r terminates depends on some hypothesis on the abstract domain such as to be a complete lattice (when this is not the case, and it is just a cpo, an additional widening operation is usually provided by the domain).

Example 2. Consider again the program PERMUTE illustrated above. Using one of our domains for abstract substitutions, like Pattern (see [5,20]), and starting from an activation state of the form **perm(ground,var)** our analysis returns the abstract qualified answer $\langle$**perm(ground, ground)**$, \nu\rangle$, which provides the information that any concrete execution, starting in a query of perm with the first argument being ground and the second one being variable, is deadlock free.

5 Conclusions

The semantics that has been discussed in these pages belongs to the foundation part of a project aimed at integrating most of the work (both theoretical and practical) on abstract interpretation of logic programs developed by the authors in the last years. The goal is to get a practical tool that tackles a variety of problems

$$TAB(sat) = \{(\beta, p, \langle\beta', \kappa\rangle) : (\beta, p) \text{ is an } \mathit{input\ state} \text{ and } \langle\beta', \kappa\rangle = T_p(\beta, p, sat)\}.$$

$$\begin{array}{ll}
T_p(\beta, p, sat) = \mathtt{UNION}(\langle\beta_1, \kappa_1\rangle \ldots, \langle\beta_n, \kappa_n\rangle) & \\
\quad\textbf{where} & \langle\beta_i, \kappa_i\rangle = T_c(\beta, c_i, sat), \\
 & c_1, \ldots, c_n \text{ are the clauses defining } p.
\end{array}$$

$$\begin{array}{ll}
T_c(\beta, c, sat) = \langle\mathtt{RESTRC}(c, \beta'), \kappa\rangle & \\
\quad\textbf{where} & \langle\beta', \kappa\rangle = T_b(\mathtt{EXTC}(c, \beta), g\_d, g\_e, <>, sat), \\
 & \langle g\_d, g\_e\rangle = \mathtt{SPLIT}(\beta, b) \text{ where } b \text{ is the body of } c.
\end{array}$$

$$T_b(\beta, <>, <>, <>, sat) = \langle\beta, \nu\rangle.$$

$$\begin{array}{ll}
T_b(\beta, g\_d, <>, g\_r, sat) = \langle\beta, \mu\rangle & \\
\quad\textbf{where} & \text{either } g\_d \text{ or } g\_r \text{ is not empty.}
\end{array}$$

$$\begin{array}{lll}
T_b(\beta, g\_d, l.g\_e, g\_r, sat) = T_b(\beta_4, \bar{g}\_d, g\_e.\bar{g}\_e, \bar{g}\_r, sat) & & \\
\quad\textbf{where} & \langle\bar{g}\_d, \bar{g}\_e\rangle = \mathtt{SPLIT}(\beta_4, g\_d) & \\
 & \langle\beta_4, \bar{g}\_r\rangle = T_r(\beta_3, g\_r, sat) & \text{if } \kappa = \nu, \\
 & \qquad\qquad\quad T_r(\beta_3, g\_r.l, sat) & \text{if } \kappa = \mu, \\
 & \beta_3 = \mathtt{EXTG}(l, \beta, \beta_2), & \\
 & \langle\beta_2, \kappa\rangle = \ sat(\beta_1, p) & \text{if } l \text{ is } p(\cdots) \\
 & \qquad\qquad \langle\mathtt{UNIF\_VAR}(\beta_1), \nu\rangle & \text{if } l \text{ is } x_i = x_j, \\
 & \qquad\qquad \langle\mathtt{UNIF\_FUNC}(l, \beta_1), \nu\rangle & \text{if } l \text{ is } x_i = f(\cdots), \\
 & \beta_1 = \mathtt{RESTRG}(l, \beta). &
\end{array}$$

$$\begin{array}{ll}
T_r(\beta, (a_1, \ldots, a_n), sat) = \sqcap_{i=1}^{\infty}\langle\beta_i, g_i\rangle & \\
\quad\textbf{where} & \langle\beta_0, g_0\rangle = \langle\beta, (a_1, \ldots, a_n)\rangle \\
 & \beta_{i+1} = \mathtt{REFINE}(\beta_i, T_r(\beta_i, a_1, sat), \ldots, T_r(\beta_i, a_n, sat))\ (i \geq 1) \\
 & g_{i+1} = \{a_i \mid i \in \{1, \ldots, n\} \text{ and } \langle\bullet, \mu\rangle = T_r(\beta_i, a_i, sat)\}
\end{array}$$

$$\begin{array}{lll}
T_r(\beta, a, sat) = \langle\mathtt{RENAME}(a, \beta_2), \kappa\rangle & & \\
\quad\textbf{where} & \langle\beta_2, k\rangle = sat(\beta_1, p) & \text{if } a \text{ is } p(\cdots) \\
 & \beta_1 = \mathtt{RESTRG}(a, \beta). &
\end{array}$$

Fig. 3. The abstract transformation

raised by the recent research and development directions in declarative programming. Dynamic scheduling is an interesting example in that respect. In the next future, we plan to adapt the existing implementations of GAIA systems in order to practically evaluate the accuracy and efficiency of these seminal ideas.

References

1. K. R. Apt. *From Logic Programming to Prolog.* Prentice Hall, 1997.
2. K. R. Apt and I. Luitjes. Verification of logic programs with delay declarations. *Lecture Notes in Computer Science*, 936:66–80, 1995.
3. M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 10(2):91–124, February 1991.
4. M. Bruynooghe, G. Janssens, A. Callebaut, and B. Demoen. Abstract interpretation: Towards the global optimization of Prolog programs. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 192–204, San Francisco, California, August 1987. Computer Society Press of the IEEE.
5. A. Cortesi, G. Filé, and W. Winsborough. Optimal groundness analysis using propositional logic. *Journal of Logic Programming*, 27(2):137–167, May 1996.
6. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combination of abstract domains for logic programming. In *Proceedings of the 21th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94)*, Portland, Oregon, January 1994.
7. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combination of abstract domains for logic programming: open product and generic pattern construction. *Science of Computer Programming*, 28(1–3):27–71, 2000.

8. A. Cortesi, S. Rossi, and B. Le Charlier. Operational semantics for reexecution-based analysis of logic programs with delay declarations. *Electronic Notes in Theoretical Computer Science*, 48(1), 2001. http://www.elsevier.nl/locate/entcs.
9. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of Fourth ACM Symposium on Programming Languages (POPL'77)*, pages 238–252, Los Angeles, California, January 1977.
10. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of Sixth ACM Symposium on Programming Languages (POPL'79)*, pages 269–282, Los Angeles, California, January 1979.
11. M. Garcia de la Banda, K. Marriott, and P. Stuckey. Efficient analysis of logic programs with dynamic scheduling. In J. Lloyd, editor, *Proc. Twelfth International Logic Programming Symposium*, pages 417–431. MIT Press, 1995.
12. M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Constraint logic programming with dynamic scheduling: A semantics based on closure operators. *Information and Computation*, 137(1):41–67, 1997.
13. Intelligent Systems Laboratory, Swedish Institute of Computer Science, PO Box 1263, S-164 29 Kista, Sweden. *SICStus Prolog User's Manual*, 1998. `http://www.sics.se/isl/sicstus/sicstus_toc.html`.
14. B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(1):35–101, January 1994.
15. B. Le Charlier and P. Van Hentenryck. Reexecution in abstract interpretation of Prolog. *Acta Informatica*, 32:209–253, 1995.
16. J.W. Lloyd. *Foundations of Logic Programming*. Springer Series: Symbolic Computation–Artificial Intelligence. Springer-Verlag, second, extended edition, 1987.
17. E. Marchiori and F. Teusink. Proving termination of logic programs with delay declarations. In John Lloyd, editor, *Proceedings of the International Symposium on Logic Programming*, pages 447–464, Cambridge, December 4–7 1995. MIT Press.
18. K. Marriott, M. Garcia de la Banda, and M. Hermenegildo. Analyzing logic programs with dynamic scheduling. In *Proc. 21st Annual ACM Symp. on Principles of Programming Languages*, pages 240–253. ACM Press, 1994.
19. L. Naish. *Negation and control in Prolog*. Number 238 in Lecture Notes in Computer Science. Springer-Verlag, New York, 1986.
20. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of the domain *Prop*. *Journal of Logic Programming*, 23(3):237–278, June 1995.
21. K. Yelick and J. Zachary. Moded type systems for logic programming. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages (POPL'89)*, pages 116–124, 1989.

Pos($\mathcal{T}$): Analyzing Dependencies in Typed Logic Programs

Maurice Bruynooghe[1], Wim Vanhoof[1], and Michael Codish[2]

[1] Katholieke Universiteit Leuven, Department of Computer Science
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
e-mail: {maurice,wimvh}@cs.kuleuven.ac.be
[2] Ben-Gurion University, Department of Computer Science,
P.O.B. 653, 84105 Beer-Sheva, Israel
e-mail: mcodish@cs.bgu.ac.il

Abstract. Dependencies play a major role in the analysis of program properties. The analysis of groundness dependencies for logic programs using the class of *positive* Boolean functions is a main applications area. Work has been done to improve its precision through the integration of either pattern information or type information. This paper develops another approach where type information is exploited. Different from previous work, a separate simple analysis is done for each subtype of the types. Also, a technique is developed that reuses the results of a polymorphic predicate for the type instances under which it is called.

1 Introduction

Dependencies play an important role in program analysis. A statement "program variable X has property p" can be represented by the propositional variable x^p and dependencies between properties of program variables can be captured as Boolean functions. For example, the function denoted by $x^p \rightarrow y^p$ specifies that whenever x has property p then so does y. In many cases, the precision of a dataflow analysis for a property p is improved if the underlying analysis domain captures dependencies with respect to that given property.

The analysis of groundness dependencies for logic programs using the class of *positive* Boolean functions is one of the main applications in this area of research. The analysis aims at identifying if program variable[1] X has a unique value which cannot be changed. In logic programming terms this means that X is *ground*, or, contains no variables which can be further instantiated. This is the property presented by the propositional variable x. The class of *positive* Boolean functions, *Pos* consists of the Boolean functions for which $f(true, \ldots, true) = true$.

One of the key steps in a groundness dependency analysis is to characterise the dependencies imposed by the unifications that could occur during execution. If the program specifies a unification of the form $term_1 = term_2$ and the variables in $term_1$ and $term_2$ are $\{X_1, \ldots, X_m\}$ and $\{Y_1, \ldots, Y_n\}$ respectively, then the corresponding groundness dependency imposed is $(x_1 \wedge \cdots \wedge x_m) \leftrightarrow (y_1 \wedge \cdots \wedge y_n)$ specifying that variables in $term_1$ are (or will become) ground if and only if the variables in $term_2$ are (or do).

It is possible to improve the precision of an analysis if additional information about the structure (or patterns) of terms is available. For example, if we know that $term_1$ and $term_2$ are both difference lists of the form $H_1 - T_1$ and $H_2 - T_2$, respectively, then the unification $term_1 = term_2$ imposes the dependency $(h_1 \leftrightarrow h_2) \wedge (t_1 \leftrightarrow t_2)$ which is more precise than $(h_1 \wedge t_1) \leftrightarrow (h_2 \wedge t_2)$ which would be derived without the additional information. This has been the approach in previous works such as [20, 24, 8, 11, 2] where simple pattern analysis can be used to enhance the precision of other analyses.

Introducing pattern information does not allow to distinguish between e.g. bounded lists such as $[1, X, 3]$ and open ended lists such as $[1, 2|Z]$ because the open end can be situated at an arbitrary depth. Making such distinction requires to consider type information. The domain *P*os has been adapted to do so in [6] where each type was associated with an incarnation of *P*os. However, that analysis was for untyped programs and each incarnation was developed in somewhat ad-hoc fashion. related. Here, the models of the program, based on different pre-interpretations express different kinds of program properties. But again, the choice of a pre-interpretation is on a case by case basis. Others in one or another way annotate the types with information about the positions where a variable can occur. This is the case in: [26, 27]; the binding time analysis of [30]; and also in [21] which uses types and applies linear refinement to enrich the type domain with *P*os-like dependencies. our approach is the work of [19] which associates properties with the subtypes of a variable. In

[1] We use upper case for program variables and lower case for the corresponding propositional variable in a formula expressing dependencies.

that work, abstract unifications are abstracted by Boolean formulas expressing the groundness dependencies between different subtypes. The main difference is that they construct a single compiled clause covering all subtypes while we construct one clause for each subtype. A feature distinguishing our work from the other type based approaches is that we have an analysis for polymorphic types which eliminates the need to analyze a polymorphic predicate for each distinct type instance under which it is called.

Type information can be derived by analysis, as e.g. in [18, 15]; specified by the user and verified by analysis as possible in Prolog systems such as Ciao [16]; or declared and considered part of the semantics of the program as with strongly typed languages such as Gödel [17], Mercury [28] and HAL [13]. The analysis as worked out in this paper is for strongly typed programs.

In the next section we recall briefly the essentials of groundness dependency analysis using *Pos*. We exemplify the simple and elegant implementation technique for the analysis based on program abstraction as described in [7]. as used in the paper as well as some relationships between types that will be used in later sections. analysis for monomorphic types, it defines the abstraction for different kinds of unification and proves that they are correct. 5 deals with the polymorphic case. It describes how to abstract a call with types that are an instance of the polymorphic types in the called predicate's definition. It proves that the results of the polymorphic analysis approximate the results of a monomorphic analysis and points out the (frequent) cases where both analyses are equivalent. Section 6 discusses applications and related work.

2 Analyzing Groundness Dependencies with *Pos*

Program analysis aims at computing finite approximations of the possibly infinite number of program states that could arise at runtime. Using abstract interpretation [12], approximations are expressed using elements of an abstract domain and are computed by abstracting a concrete semantics; the algebraic properties of the abstract domain guarantee that the analysis is terminating and correct.

The formal definition of *Pos* states that a *Pos* function φ describes a substitution θ (a program state) if any set of variables that might become ground by further instantiating θ is a model of φ. For example, the models of $\varphi = x \wedge (y \rightarrow z)$ are $\{\{X\}, \{X, Z\}, \{X, Y, Z\}\}$. We can see that φ describes $\theta = \{X/a, Y/f(U, V)), Z/g(U)\}$ because under further instantiation X is in all of the models and if Y is in a model (becomes ground) then so is Z. Notice that $\theta = \{X/a\}$ is not described by φ as $\{X/a, Y/a\}$ is a further instantiation of θ and $\{X, Y\}$ is not a model of φ. of an abstract domain [9]. [23] for more details.

A simple way of implementing a *Pos* based groundness analysis is described in [7] and is illustrated in Figure 1. For the purpose of this paper it is sufficient to understand that the problem of analyzing the concrete program (on the left part of Figure 1) is reduced to the problem of computing the concrete semantics of the abstract program (in the middle and on the right). The result is given at the bottom of the figure. For additional details of why this is so, refer to [7, 10, 23]. The least model of the abstract program (e.g. computed using meta-interpreters such as those described in [5, 7]) is interpreted as representing the propositional formula $x_1 \leftrightarrow x_2$ and $(x_1 \wedge x_2) \leftrightarrow x_3$ for the atoms $\texttt{rotate}(X_1, X_2)$ and $\texttt{append}(X_1, X_2, X_3)$ respectively. This illustrates a goal-independent analysis. Goal-dependent analyses are supported by applying Magic sets or similar techniques (see e.g. [7]).

3 About Terms and Types

We assume familiarity with logic programming concepts [22, 1]. We let $T(\Sigma, V)$ denote the set of terms constructed from a set of function symbols Σ and variables V. Substitutions are mappings from V to $T(\Sigma, V)$ and defined as usual.

We assume a standard notion of strong typing as for example in Mercury [28]. and (type) symbols. We denote by $\mathcal{T}(\Sigma_\mathcal{T}, V_\mathcal{T})$ the set of types constructed from type variables $V_\mathcal{T}$ and type symbols $\Sigma_\mathcal{T}$. *polymorphic,* otherwise it is *monomorphic.* Type substitutions are substitutions from type variables to types. The application of a type substitution to a polymorphic type gives a new type which is an *instance* of the original type.

Function and type symbols are associated with an arity. We write $f/n \in \Sigma$ (or $f/n \in \Sigma_\mathcal{T}$) to specify that f is an n-ary symbol. We assume that the sets of symbols, variables, type symbols and type variables are fixed, $\Sigma \cap \Sigma_\mathcal{T} = \emptyset$, $V \cap V_\mathcal{T} = \emptyset$, and use *Term* for $T(\Sigma, V)$ and $\mathcal{T}$ for $\mathcal{T}(\Sigma_\mathcal{T}, V_\mathcal{T})$. We use $t : \tau$ to denote that term t has type τ.

We restrict our attention to *well-typed* terms and substitutions. The relation between types and the terms belonging to them is made explicit by a type definition which consists of a finite set of type rules. For each type symbol, a unique type rule associates that symbol with a finite set of function symbols.

Concrete `rotate`	*Abstract* `rotate`	*Auxiliary predicates*

`rotate(Xs,Ys) :-`
`    append(As,Bs,Xs),`
`    append(Bs,As,Ys).`
	`rotate(Xs,Ys) :-`
	`    append(As,Bs,Xs),`
	`    append(Bs,As,Ys).`
		`iff(true,[]).`
		`iff(true,[true|Xs]) :-`
		`    iff(true,Xs).`
		`iff(false,Xs) :-`
		`    member(false,Xs).`

`append(Xs,Ys,Zs) :-`
`    Xs = [],`
`    Ys = Zs.`
	`append(Xs,Ys,Zs) :-`
	`    iff(Xs,[]),`
	`    iff(Ys,[Zs]).`	

`append(Xs,Ys,Zs) :-`
`    Xs = [X|Xs1],`
`    Zs = [X|Zs1],`
`    append(Xs1,Ys,Zs1).`
	`append(Xs,Ys,Zs) :-`
	`    iff(Xs,[X, Xs1]),`
	`    iff(Zs,[X, Zs1]),`
	`    append(Xs1,Ys,Zs1).`	

Least model (abstract `rotate`*):*	`rotate(X,X).`	`append(true,true,true).`
		`append(false,Y,false).`
		`append(X,false,false).`

Fig. 1. Concrete and abstract programs; least model of the abstract program.

Definition 1. *The rule for a type symbol $h/n \in \Sigma_\mathcal{T}$ is a definition of the form*

$$h(\bar{V}) \longrightarrow f_1(\bar{\tau}_1)\,;\ldots\,;\,f_k(\bar{\tau}_k).$$

where: $\bar{V}$ is an n-tuple from $V_\mathcal{T}$; for $1 \leq i \leq k$, $f_i/m \in \Sigma$ with $\bar{\tau}_i$ an m-tuple from $\mathcal{T}$; and type variables occurring in the right hand side occur in the left hand side as well[2]*. The function symbols $\{f_1,\ldots,f_k\}$ are said to be associated with the type symbol h. A finite set of type rules is called a type definition.*

Given the type of a (possibly non-ground) term, one can derive a type for every subterm of the term, in particular for the variables occurring in the term.

Example 1. Consider the following type rule introduced using the keyword `type`:

```
type  list(T) ---> [] ; [T | list(T)].
```

The function symbols [] (nil) and | (cons) are associated with the type symbol *list*. The type definition defines also the denotation of each type (the set of terms belonging to the type). For this example, type $list(T)$ are either variables (typed $list(T)$, of the form [], or of the form $[t_1|t_2]$ with t_1 of type T and t_2 of type $list(T)$. its structure under instantiation. Hence only a variable (typed T) can be of type T. Applying the type substitution $\{T/int\}$ on $list(T)$ gives the type $list(int)$. Terms of the form $[t_1|t_2]$ are of type $list(int)$ if t_1 is of type int and t_2 is of type $list(int)$. Type instances can also be polymorphic, e.g. $list(list(T))$.

The next definition specifies the *constituents* of a type τ. These are the types of the terms out of which a term of type τ is constructed, in other words, the possible types of the subterms of a term of type τ.

Definition 2. *The constituents relation for type definition ρ is the minimal pre-order (reflexive and transitive) $\preceq_\rho\colon \mathcal{T} \times \mathcal{T}$ such that if $h(\bar{\tau}) \longrightarrow f_1(\bar{\tau}_1)\,;\ldots\,;\,f_k(\bar{\tau}_k)$ is an instance of a rule in ρ and τ is an argument of $f_i(\bar{\tau}_i)$, then $\tau \preceq_\rho h(\bar{\tau})$. The set of constituents of a type τ is $Constituents_\rho(\tau) = \left\{\, \tau' \in \mathcal{T} \,\middle|\, \tau' \preceq_\rho \tau \,\right\}$. When ρ is clear from the context we omit it in the notation for $\preceq_\rho$ and $Constituents_\rho$.*

Example 2. With $list/1$ as in Example 1 and the atomic type int, we have:

- $T \preceq list(T)$, $list(T) \preceq list(T)$, $int \preceq list(int)$, $int \preceq int$, $list(T) \preceq list(list(T))$, $T \preceq list(list(T))$, $list(list(T)) \preceq list(list(T))$, $T \preceq T$, ...
- $Constituents(int) = \{int\}$, $Constituents(T) = \{T\}$, $Constituents(list(T)) = \{T, list(T)\}$, $Constituents(list(int)) = \{int, list(int)\}$.

Next we define an instantiation property on terms.

Definition 3. *Let τ and τ' be types in a type definition ρ. We say that a term $t : \tau'$ is instantiated with respect to the type τ if there does not exist a well-typed instance $t\sigma : \tau'$ containing a variable of type τ. The predicate $\mu_\tau^\rho(t)$ is true if and only if t is instantiated with respect to type τ defined in ρ.*

[2] Types should also be well-defined: the right hand side should not use directly or indirectly an instance of $h(\bar{V})$.

The values of $\mu^{\rho}_{list(int)}(s)$ and $\mu^{\rho}_{int}(s)$ for some terms s of type *list(int)* are:

s	$\mu_{list(int)}(s)$	$\mu_{int}(s)$
$[1,2]$	*true*	*true*
$[1,X]$	*true*	*false*
$[1\|X]$	*false*	*false*

For instance, $\mu_{list(int)}([1,X]) = \mathit{true}$ because all *list(int)*-subterms of well-typed instances of $[1,X]$ are instantiated. On the other hand, $\mu_{list(int)}([1|X]) = \mathit{false}$ because the subterm X of type *list(int)* is a variable. Also $\mu_{int}([1|X]) = \mathit{false}$ as e.g. $[1,Y|Z]$ is an instance with the variable Y of type *int*. Classical groundness can be defined in terms of μ^{ρ}_{τ}. For a term t of type τ:

$$\mathit{ground}(t) \leftrightarrow \wedge \{ \, \mu^{\rho}_{\tau_i}(t) \, | \, \tau_i \in \mathit{Constituents}(\tau) \, \} \, . \tag{1}$$

4 *Pos*($\mathcal{T}$) in a Monomorphic Setting

In what follows, we write x^{τ} to indicate that x is a propositional variable about type τ. In *Pos*($\mathcal{T}$) analysis, the truth of propositional variable x^{τ} expresses the property of program variable X that **no instance of its value contains a variable of type** τ[3]. Hence the concretisation function for *Pos*($\mathcal{T}$) is:

Definition 4. *Let V be a set of (typed) variables of interest, τ a type and φ a positive Boolean function over $W = \{X \in V \,|\, X : \tau', \ \tau \preceq \tau'\}$. The concretisation of φ with respect to τ, denoted $\gamma_{\tau}(\varphi)$, is the set of well-typed substitutions θ such that $\{ \ X \in W \, \big| \, \mu_{\tau}(X\theta) \ \}$ is a model of φ.*

A variable $X_i : \tau_i$ is excluded from the domain of φ when τ is not a constituent of τ_i. With s_i the value of such X_i, it is the case that $\mu_{\tau}(s_i)$ is trivially true, hence instead of excluding X_i, one could state that x_i^{τ} holds. However, this causes problems for the handling of polymorphism worked out in the next section. Indeed, while τ cannot be a constituent of a polymorphic parameter T, it can be a constituent of an instance of T, hence x_i^{τ} is not necessarily true for instances.

In a classic groundness analysis, the unification $A = [X|Xs]$ is abstracted as $a \leftrightarrow (x \wedge xs)$. Indeed, we assume that any subterm of the term that A is bound to at runtime could unify with any subterm of the terms bound to X or Xs. In the presence of types we know that A and $[X|Xs]$ are both of the same type fail). (otherwise the program is not well-typed). In addition, we know that all unifications between subterms of (the terms bound to) A and $[X|Xs]$ are between terms corresponding to the same types. So in this example (assuming both terms to be of type *list(int)*), we can specify $a \leftrightarrow xs$ for type *list(int)* and $a \leftrightarrow (x \wedge xs)$ for type *int*. It is important to note that the interpretations of the variables in $a \leftrightarrow xs$ and in $a \leftrightarrow (x \wedge xs)$ are different. The former refers to subterms of type *list(int)* whereas the latter refers to subterms of type *int*. These intuitions are formalised in the following definitions and theorems.

Definition 5. *τ-abstraction I*
Let τ be a type and X, Y program variables of type τ'. The τ-abstraction of $X = Y$ is: if $\tau \notin \mathit{Constituents}(\tau')$ then true else $x^{\tau} \leftrightarrow y^{\tau}$.

Theorem 1. *Correctness.*
Let φ a positive Boolean function on a set of variables V including X and Y and φ' the τ-abstraction of $X = Y$. If $\theta \in \gamma_{\tau}(\varphi)$ and $\sigma = mgu(X\theta, Y\theta)$ then $\theta\sigma|_V \in \gamma_{\tau}(\varphi \wedge \varphi')$.

Definition 6. *τ-abstraction II*
Let τ be a type, $X, Y_1, \ldots, Y_n$ variables of types $\tau_0, \tau_1, \ldots, \tau_n$ respectively. The τ-abstraction of $X = f(Y_1, \ldots, Y_n)$ is: if $\tau \notin \mathit{Constituents}(\tau_0)$ then true else[4]

$$x^{\tau} \leftrightarrow \wedge \{ \, y_i^{\tau} \, | \, 1 \leq i \leq n, \ \tau \preceq \tau_i \, \} \, . \tag{2}$$

Theorem 2. *Correctness.*
Let φ be a positive Boolean function on a set of variables V including $X, Y_1, \ldots, Y_n$ and φ' the τ-abstraction of $X = f(Y_1, \ldots, Y_n)$. If $\theta \in \gamma_{\tau}(\varphi)$ and $\sigma = mgu(X = f(Y_1, \ldots, Y_n)$ then $\theta\sigma|_V \in \gamma_{\tau}(\varphi \wedge \varphi')$.

[3] This is a generalisation of *Pos* where x expresses that no instance of the value contains a variable, i.e. the value is ground.

[4] It reduces to $x \leftrightarrow \mathit{true}$ when τ is not a constituent of any of the types $\tau_1, \ldots, \tau_n$.

Having shown that the τ-abstraction of unification is correct, we can rely on the results of [7] for the abstraction of the complete program, for the computation of its least model and for the claim that correct answers to a query $\leftarrow p(X_1,\ldots,X_n)$ belong to the concretisation of the $Pos(\mathcal{T})$-formula of the predicate p/n. Basic intuitions are that the $Pos(\mathcal{T})$ formula of a single clause consists of the conjunction of the $Pos(\mathcal{T})$ formulas of the body atoms and that the $Pos(\mathcal{T})$ formula of a predicate consists of the disjunction (lub) of the $Pos(\mathcal{T})$ formulas of the individual clauses defining the predicate.

A simple implementation technique, replacing unifications by appropriate calls to `iff/2` (as in Section 2 for Pos) is illustrated for append/3 in Figure 2.

Abstraction for type constituent list(int)	*Abstraction for type constituent int*
append_list_int(Xs,Ys,Zs) :-	append_int(Xs,Ys,Zs) :-
iff(Xs,[]),	iff(Xs,[]),
iff(Ys,[Zs]).	iff(Ys,[Zs]).
append_list_int(Xs,Ys,Zs) :-	append_int(Xs,Ys,Zs) :-
iff(Xs,[Xs1]),	iff(Xs,[X, Xs1]),
iff(Zs,[Zs1],	iff(Zs,[X, Zs1]),
append_list_int(Xs1,Ys,Zs1).	append_int(Xs1,Ys,Zs1).

Fig. 2. Abstraction for the types in append

The least model of append_int/3 expresses the $Pos(int)$-formula $z \leftrightarrow x \wedge y$, i.e. that all subterms of Z of the type *int* are instantiated iff those of X and Y are. The least model of append_list_int/3 expresses the $Pos(list(int))$-formula $x \wedge (y \leftrightarrow z)$, i.e. that all subterms of X of type $list(int)$ are instantiated (in other words the backbone of the list is instantiated when append/3 succeeds) and that those of Y are instantiated iff those of Z are instantiated. Classical groundness, is obtained by composing the two models, using Equation (1) in Section 3:

```
append(X,Y,Z) :- append_list_int(X1,Y1,Z1), append_int(Xe,Ye,Ze),
                 iff(X,[X1,Xe]), iff(Y,[Y1,Ye]), iff(Z,[Z1,Ze]).
```

5 Polymorphism in $Pos(\mathcal{T})$

Type polymorphism is an important abstraction tool: a predicate defined with arguments of a polymorphic type can be called with actual arguments of any type that is an instance of the defined type. For example, the append/3 predicate from Fig. 2 usually is defined with respect to a polymorphic type definition, stating that each of its arguments is of type $list(T)$. Abstracting append/3 for this type definition results in the same abstractions as in Figure 2 but with constituent $list(T)$ replacing $list(int)$ and T replacing *int*.

When abstracting a call to such a predicate, one needs the abstractions with respect to the constituents of the actual types of the call (e.g., *char* and $list(char)$ in case append/3 is called with actual types $list(char)$). One possibility to obtain these, is to analyze the definition for each type-instance by which it is called. However, it is much more efficient to analyze the definition once for its given types, and derive the abstractions of a particular call from that result.

The need for such an approach is even more urgent when analyzing large programs distributed over many modules. It is preferable that an analysis does not need the actual code of the predicates it imports (and of the predicates called directly or indirectly by the imported predicates) but only the result of the call independent analysis. See [25, 4, 29] for discussions about module based analysis.

Space constraints prevent us from completely developping the polymorphic case. We only sketch the main ideas using an example. Consider the following program for a predicate `p(list(list(int)),list(list(int)))`:

Concrete definition	*Abstraction*
p(X,Y):- append(X,X,Y).	p_list_list_int(X,Y) :- append_list_list_int(X,X,Y).
	p_list_int(X,Y) :- append_list_int(X,X,Y).
	p_int(X,Y) :- append_int(X,X,Y).

Instead of analyzing append/3 for all three constituents of the type *list(list(int))*, it is preferable to reuse the results for the constituents *list(T)* and *T* of a polymorphic analysis. Doings so requires to recognize that the constituent *list(list(int))* corresponds to *list(T)* and both constituents *list(int)* and *int* to *T*. Hence one can abstract the program as:

```
p_list_list_int(X,Y) :- append_list_T(X,X,Y).
p_list_int(X,Y) :- append_T(X,X,Y).
p_int(X,Y) :- append_T(X,X,Y).
```

Things get more involved, and some precision loss may result when the constituent for which an analysis is done is a constituent of both the polymorpic type and the type instances of polymorphic type parameters. Consider the predicate *q/4* and its abstractions for the constituents *int* and *T*:

Concrete	*Abstraction w.r.t. int*	*Abstraction w.r.t.* T

`pred q(int,int,T,T).`
`q(X,Y,U,V):- X=Y,U=V.`
`q(X,Y,U,V):- X=0.`
	`q_int(X,Y,U,V):-`
	`iff(X,[Y]).`
	`q_int(X,Y,U,V):-`
	`iff(X,[]).`
		`q_T(X,Y,U,V):-`
		`iff(U,[V]).`
		`q_T(X,Y,U,V).`

Now, assume `q(A,B,C,D)` is called with A, B, C and D of type *int* and need to be analyzed for the constituent *int*. Type *int* in the call corresponds to both the constituent *int* and T of the polymorphic predicate q/4. A correct *int*-abstraction of the *call* should call both the *int*-abstraction and the T-abstraction of the polymorphic q/4. Hence, the call is abstracted as:

```
q_int(A,B,C,D), q_T(A,B,C,D).
```

6 Discussion

The main goal of this research is to improve the precision of groundness dependencies by taking into consideration type information. Precision is improved for two reasons: (a) computation paths which unify terms of different types can be pruned; and (b) when unifying terms (or variables) of the same type it is possible to refine the resulting dependecies (e.g. list elements with list elements, and list backbone with list backbone).

While, *Pos(T)* can improve the precision of groundness dependencies it is important also to note that the *Pos(T)* formulas provide valuable information on their own. In future work we intend to use it to improve automated termination analysis which in turn will be used in a binding-time analysis for the off-line specialisation of logic programs (improving the work in [3].).

The full version of this paper will contain the proofs, will develop the details of the polymorphic analysis and will also explain how the results of the analysis for some type can be used to speed-up the analysis of its constituent types.

References

1. K. R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics*, pages 493–574. Elsevier Science Publishers B.V., 1990.
2. R. Bagnara, P. M. Hill, and E. Zaffanella. Efficient structural information analysis for real CLP languages. In *Proc. LPAR2000*, volume 1955 of *LNCS*. Springer, 2000.
3. M. Bruynooghe, M. Leuschel, and K. Sagonas. A polyvariant binding-time analysis for off-line partial deduction. In *Proc. ESOP'98*, volume 1381 of *LNCS*, pages 27–41. Springer, 1998.
4. F. Bueno, M. de la Banda, M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey. A model for inter-module analysis and optimizing compilation. In *Preproceedings LOPSTR2000*, 2000.
5. M. Codish. Efficient goal directed bottom-up evaluation of logic programs. *J. Logic Programming*, 38(3):354–370, 1999.
6. M. Codish and B. Demoen. Deriving polymorphic type dependencies for logic programs using multiple incarnations of prop. In *Proc. SAS'94*, volume 864 of *LNCS*, pages 281–296. Springer, 1994.
7. M. Codish and B. Demoen. Analyzing logic programs using "PROP"-ositional logic programs and a magic wand. *J. Logic Programming*, 25(3):249–274, 1995.
8. M. Codish, K .Marriott, and C. Taboch. Improving program analyses by structure untupling,. *Journal Logic Programming*, 43(3), June 2000.
9. A. Cortesi, G. Filé, and W. Winsborough. Prop revisited: propositional formula as abstract domain for groundness analysis. In *Proc. LICS'91*, pages 322–327. IEEE Press, 1991.
10. A. Cortesi, G. Filé, and W. H. Winsborough. Optimal groundness analysis using propositional logic. *J. Logic Programming*, 27(2):137–167, 1996.
11. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of abstract domains for logic programming: open product and generic pattern construction. *Science of Computer Programming*, 38(1-3):27–71, 2000.

12. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. POPL'97*, pages 238–252, 1977.
13. B. Demoen, M. García de la Banda, W. Harvey, K. Mariott, and P. Stuckey. An overview of HAL. In *Proc. CP'99*, volume 1713 of *LNCS*, pages 174–188. Springer, 1999.
14. J. Gallagher, D. Boulanger, and H. Saglam. Practical model-based static analysis for definite logic programs. In *Proc. ILPS'95*, pages 351–365. MIT Press, 1995.
15. J. Gallagher and D. A. de Waal. Fast and precise regular approximations of logic programs. In *Proc. ICLP'94*, pages 599–613. MIT Press, 1994.
16. M. Hermenegildo, F. Bueno, G. Puebla, and P. López. Debugging, and optimization using the Ciao system preprocessor. In *Proc. ICLP'99*, pages 52–66, 1999.
17. P. Hill and J. Lloyd. *The Gödel Language.* MIT Press, 1994.
18. G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *J. Logic programming*, 13(2&3):205–258, 1992.
19. V. Lagoon and P. J. Stuckey. A framework for analysis of typed logic programs. In *Proc. FLOPS 2001*, volume 2024 of *LNCS*, pages 296–310. Springer, 2001.
20. B. Le Charlier and P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. *ACM TOPLAS*, 16(1):35–101, 1994.
21. G. Levi and F. Spoto. An experiment in domain refinement: Type domains and type representations for logic programs. In *Proc. PLILP/ALP'98*, volume 1490 of *LNCS*, pages 152–169. Springer, 1998.
22. J.W. Lloyd. *Foundation of Logic Programming.* Springer, second edition, 1987.
23. K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems*, 2(1-4):181–196, 1993.
24. A. Mulkers, W. Simoens, G. Janssens, and M. Bruynooghe. On the practicality of abstract equation systems. In *Proc. ICLP'95*, pages 781–795. MIT Press, 1995.
25. G. Puebla and M. Hermenegildo. Some issues in analysis and specialization of modular Ciao-Prolog programs. In *Proc. ICLP Workshop on Optimization and Implementation of Declarative Languages*, 1999.
26. O. Ridoux, P. Boizumault, and F. Malésieux. Typed static analysis: Application to groundness analysis of Prolog and lambda-Prolog. In *Proc. FLOPS'99*, volume 1722 of *LNCS*, pages 267–283. Springer, 1999.
27. J.G. Smaus, P. M. Hill, and A. King. Mode analysis domains for typed logic programs. In *Proc. LOPSTR'99*, volume 1817 of *LNCS*, pages 82–101. Springer, 2000.
28. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *J. Logic Programming*, 29(1–3):17–64, October–November 1996.
29. W. Vanhoof. Binding-time analysis by constraint solving: a modular and higher-order approach for Mercury. In *Proc. LPAR2000*, volume 1955 of *LNCS*, pages 399 – 416, Springer, 2000.
30. W. Vanhoof and M. Bruynooghe. Binding-time analysis for Mercury. In *Proc. ICLP'99*, pages 500 – 514. MIT Press, 1999.

A Prolog Tailoring Technique on an Epilog Tailored Procedure

[1]Yoon-Chan Jhi, [1]Ki-Chang Kim, [2]Kemal Ebcioglu

[1]Dept. of Computer Science & Engineering, Inha University
kchang@inha.ac.kr
[2]IBM T.J. Watson Research Center, Yorktown Heights

Prolog tailoring technique, an optimization method to improve the execution speed of a procedure, is proposed in this paper. When a procedure is repeatedly called and the machine has a lot of callee-saved registers, optimizing prolog and epilog of the procedure can become an important step of optimization. Epilog tailoring supported by IBM xlc compiler has been known to improve a procedure's execution speed by reducing the number of register-restoring instructions on exit points. In this paper, we propose a prolog tailoring technique that can reduce register-saving instructions at entry points. We can optimize prolog by providing multiple tailored versions of it on different execution paths of the procedure and by delaying the generation of register-saving instructions as late as possible on each path. However, generating prolog inside diamond structures or loop structures will cause incorrectness or unnecessary code repetition. We propose a technique to generate efficient prolog without such problems based on Tarjan's algorithms to detect SCCs(Strongly Connected Components) and BCCs (Bi-Connected Components).

1 Introduction

In a procedure call, some registers, called callee-saved registers, should preserve their values across the call; that is their values should be the same before and after the call. The callee guarantees it by saving those registers before starting the actual function body and restoring them later before leaving the code [2,3]. The register-saving instructions are called a prolog, while the register-restoring ones called an epilog. Every time this procedure is called, the prolog and epilog should be executed. For frequently executed procedures, therefore, they consume significant amount of time and are an important source of optimization [4,5].

In order to reduce the overhead of prolog and epilog code, the traditional technique was to compute those callee-saved registers that are actually killed inside the procedure. They are, then, saved and later restored in the prolog and epilog code, respectively. In this paper, we propose techniques to further reduce the number of registers that need to be saved and restored by tailoring the prolog and epilog to different execution paths. We observe that if the procedure has several execution paths, and if each path is modifying different sets of callee-saved registers, then, we may provide a different pair of prolog and epilog for each path. Since they are saving and restoring only those registers that are killed in the particular path, we can reduce the size of them.

Tailoring epilog has been implemented in some compilers, e.g. IBM xlc compiler, and the algorithm is explained in [8]. In [5], a brief mention on prolog tailoring has been made, but detailed algorithm is not published yet. In this paper, we provide the detailed algorithm and examples of prolog tailoring. The paper is organized as follows. Section 2 explains the existing epilog tailoring technique and some related researches. Section 3 explains the basic idea of the proposed prolog tailoring technique. Section 4 describes the proposed prolog tailoring algorithm in detail. Section 5 and 6 gives out experimental results and a conclusion.

2 Epilog Tailoring and Related Researches

Epilog tailoring tries to minimize the number of register-restoring operations at each exit point. The basic technique is to split the exit point. By splitting it, the set of killed registers (therefore the set of should-be-restored registers) can be different at different exit points, and we can restore only those actually killed registers at each exit point.

Fig. 1 shows an example. Fig. 1(a) is the prolog and epilog code generated without any tailoring process. In the procedure, r28, r29, r30, and r31 are killed; therefore they are saved and restored at the entrance and exit points. Fig. 1(b) shows the same procedure with tailored epilog code. The original exit point is split into two: e1 and e2 in the figure. At the paths reaching e1, the first exit point, r28 and r30 are killed, while at the path reaching e2, r29 and r31 are killed. Therefore we can have a different (and a smaller) epilog code at each exit point. The second exit point, e2, may be split into two again, optimizing the epilog code further. The procedure in Fig. 1(a) will execute 4+4 register saving/restoring operations, while that in Fig. 1(b) will execute 4+2 register saving/restoring operations regardless of which exit point it takes.

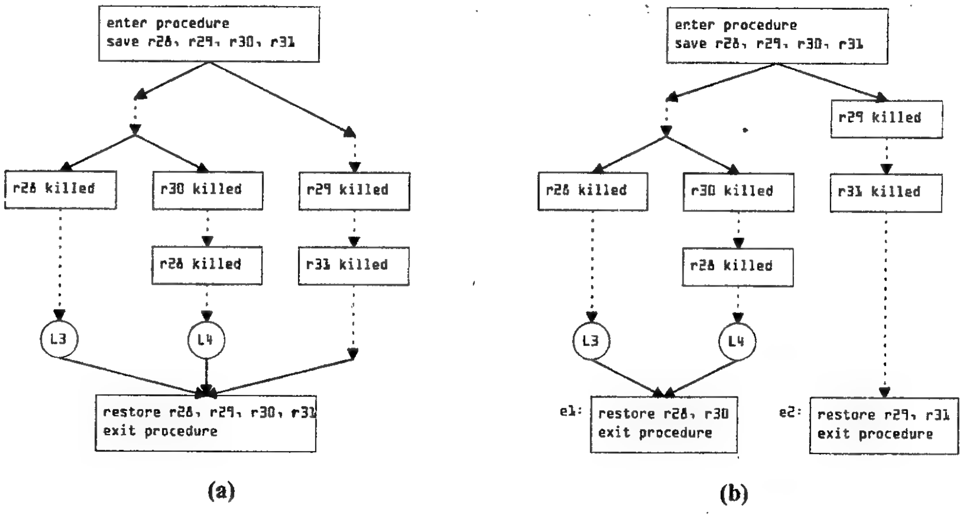


Fig. 1. Applying epilog tailoring technique on a procedure.

Other efforts to optimize prolog/epilog of procedures have been reported in [5,6,7,10]. In [5], Huang investigates the reuse of output results of some basic blocks during the run time when the same input values to them are detected. Not all basic blocks are reusable, because the input values are rarely identical for different executions of the basic blocks. But a prolog basic block is a good candidate for such reusing technique, because a procedure is often called with the same parameters. In [6,7,10], the output reusing is reduced to a single instruction. Prolog and epilog again provide a good source of instructions for such technique. Both cases do not reduce the absolute number of prolog/epilog instructions as in our case.

3 Prolog Tailoring

The basic idea of prolog tailoring is to push down the location of the register-saving operations along the execution paths as close as possible to the point where the registers are actually killed. Fig. 2 shows how prolog codes are generated on the same code as in Fig. 1(b). It saves only 2 registers at all entrance points, while the code in Fig. 1(b) saves 4. As the result, regardless of which path the procedure takes in the run time, the code in Fig. 1(b) expends 6 operations in register saving/restoring, while the code in Fig. 2 expends 4 operations.

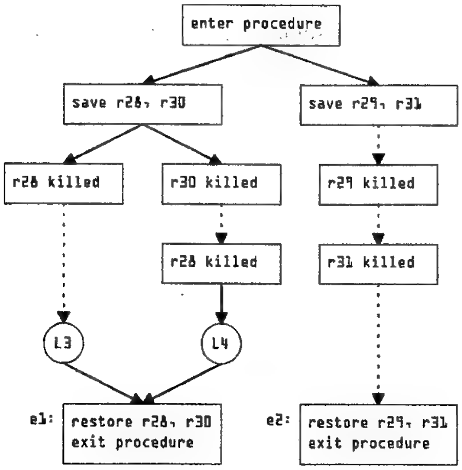


Fig. 2. Applying prolog tailoring technique on an epilog tailored procedure.

One important question in prolog tailoring is how far we can push down the register-saving operations. If a basic block is killing register r1, the saving operation for r1 can be pushed down to the entrance point of the basic block. If a register is killed in several basic blocks that have a common parent node, its corresponding saving operation can move down to the entrance point of the basic block where the parent node belongs to. Moving it further down will cause duplication of register-saving operations. If a register is killed inside a loop, the corresponding saving operation should be stopped before the loop. Once entering the loop, the register-saving operation will be executed repeatedly, wasting the CPU time. Finally, if a register is killed inside a diamond structure, e.g. if-then-else structure, the corresponding saving operation should be located before the structure, unless the join point of this diamond is split.

Pushing register-saving operations inside a diamond structure may modify the semantics of the original code. Fig. 3 shows an example. In the figure, we have pushed down the register-saving operations into a diamond structure to make them closer to the destruction points. The path reaching L3 kills only r28, while the path reaching L4 kills r28 and r30. Therefore, the code in Fig. 3 saves r28 on L3 path and r28 and r30 on L4 path. However, at exit 1, the jointing point of L3 and L4 path, r28 and r30 both are restored. If the program took L3 path during the run time, we are saving r28 only and restoring r28 and r30. Since the stack frame does not contain the original value of r30, the final value restored in r30 becomes unpredictable.

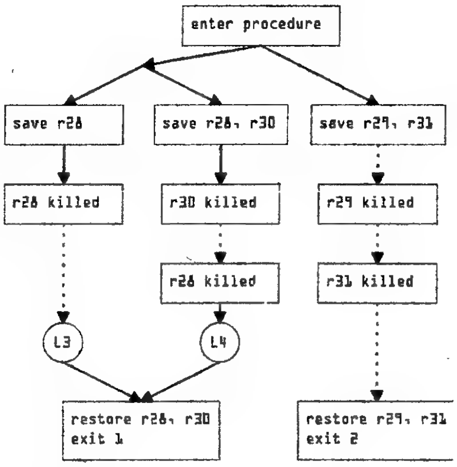


Fig. 3. Register-saving code generated inside a diamond structure.

In this paper, we propose algorithms to push down register-saving operations as close as possible to the actual destruction points but not with unnecessary duplication nor with incorrect modification of the original program's semantics.

4. Prolog Tailoring Algorithm

We assume a control glow graph is already built for a procedure for which we want to add prolog and epilog. Further, we assume the epilog is already tailored as explained in Section 2. The first step to tailor the prolog is to detect diamond and loop structures and replace them with a single node. With this replacement, the control flow graph will become a tree. On this tree, we compute DKR (Definitely Killed Registers) at each node and determine which register should be saved where, based on these DKRs. The overall algorithm is in Fig. 4, and its major steps are explained in the following sections.

```
basicblockfg_t generate_prolog(basicblockfg_t bbfg)
{
  sccfg ← remove loops from basic block flow graph;
  bccfg ← remove diamond structures from sccfg;
  dkr ← compute DKR(Definitely Killed Register) for each node in bccfg;
  tailored_bbfg ← generate register-saving operations on bbfg based on dkr and bccfg;
  return tailored_bbfg ;
}
```

Fig. 4. Basic steps of prolog tailoring.

4.1. Removing Loops

The first step of prolog tailoring is to remove loops. Loops can be identified as SCCs (Strongly Connected Components), and we can use Tarjan's algorithm [9] to detect them. Fig. 5 shows how a loop is replaced with a single node in a control flow graph. Node 2, 3, and 4 in Fig. 4(a) form an SCC; they are replaced with a single node as in Fig. 4(b). All edges reaching node 2, 3, and 4 should also reach the new replaced node, and all leaving edges from them should also leave from the new node. The new graph with all loops removed is called an SCC flow graph.

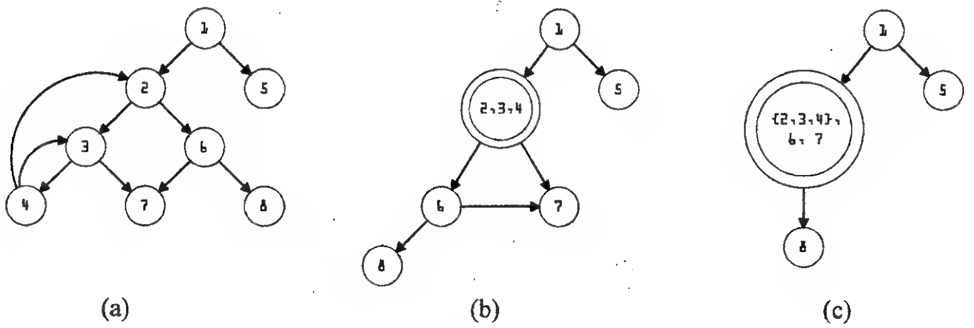


Fig. 5. Constructing SCC flow graph and BCC flow graph.

4.2. Removing Diamond Structures

The second step is to remove all diamond structures on the SCC flow graph. The modified graph is called a BCC flow graph. We can detect diamond structures by detecting BCCs (Bi-Conected Components) [1,9]. To define a BCC, let's define a bi-connected graph and an articulation point as in [1]. An articulation point is a node in a graph that divides the graph into two or more sub-graphs when it is removed. A bi-connected graph is one that does not have any articulation point. A BCC is a bi-connected sub-graph inside a full graph. Node {2,3,4}, 6, and 7 in Fig. 5(b) form a BCC; therefore they form a diamond structure. By replacing them with a single node, we get Fig. 5(c).

The BCCs detected by Tarjan's algorithm may contain shared nodes, nodes contained in more than one BCC. We need a systematic way to decide the membership of such a shared node.

Table 1. BCC set found in Fig. 5(b).

BCC	1	2	3	4
SCC	1, {2, 3, 4}	1, 5	{2, 3, 4}, 6, 7	6, 8

Table 1 shows the four BCCs found in Fig. 5 by Tarjan's algorithm. In the table, node 6 belongs to BCC node 3 and 4; therefore it is a shared node. The algorithm to remove shared nodes is in Fig. 6. In the algorithm, a local root node of a BCC is an entrance node to that BCC. The overall algorithm to obtain a BCC flow graph from an SCC flow graph is in Fig. 7.

```
bccset_t remove_shared_node(bccset_t bccset)
{
   for (all BCCs in bccset)
      if (its local root nodes has outgoing edges from this BCC)
        remove this local root node;
   for (all BCCs in bccset)
      if (there is a shared node){
           remove the shared node in the parent BCC;
   if (the parent BCC becomes empty)
     remove the parent bcc from bccset;
      }
   if (the root of sccfg was removed){
     generate a BCC that includes the root of sccfg as the only member;
        add this BCC into bccset;
   }
   return bccset ;
}
```

Fig. 6. Algorithm for shared node removal in a given BCC set.

```
bccfg_t scc_to_bcc(sccfg_t sccfg)
{
   bccset ← detect all BCCs from sccfg;
   bccset ← remove_shared_node(bccset) ;
   bccfg ← add links among BCCs in bccset based on the edges in sccfg;
   return bccfg ;
}
```

Fig. 7. Algorithm for constructing a BCC flow graph from a given SCC flow graph.

4.3. Computing DKRs

The third step is to compute killed registers at each node in the BCC flow graph and to compute DKRs based on them. A DKR of a BCC node represents a set of registers that are definitely killed in all paths starting from this BCC node. Fig. 8 shows a BCC flow graph with killed-registers and a DKR at each node. For example, at node 1, we can see r27 is killed inside node 1, and r28 is killed at both paths starting from node 1; therefore the DKR for node 1 includes r27 and r28. The DKR of node n can be defined recursively as follows.

$$DKR(n) = \bigcap_{\text{for } j \in child(n)} DKR(j) + killed\_reg(n)$$

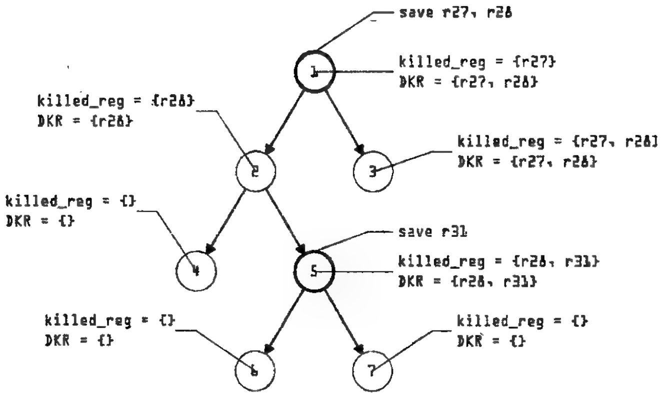


Fig 8. Computing DKR of each node and generating register-saving instructions.

4.4. Prolog Generation

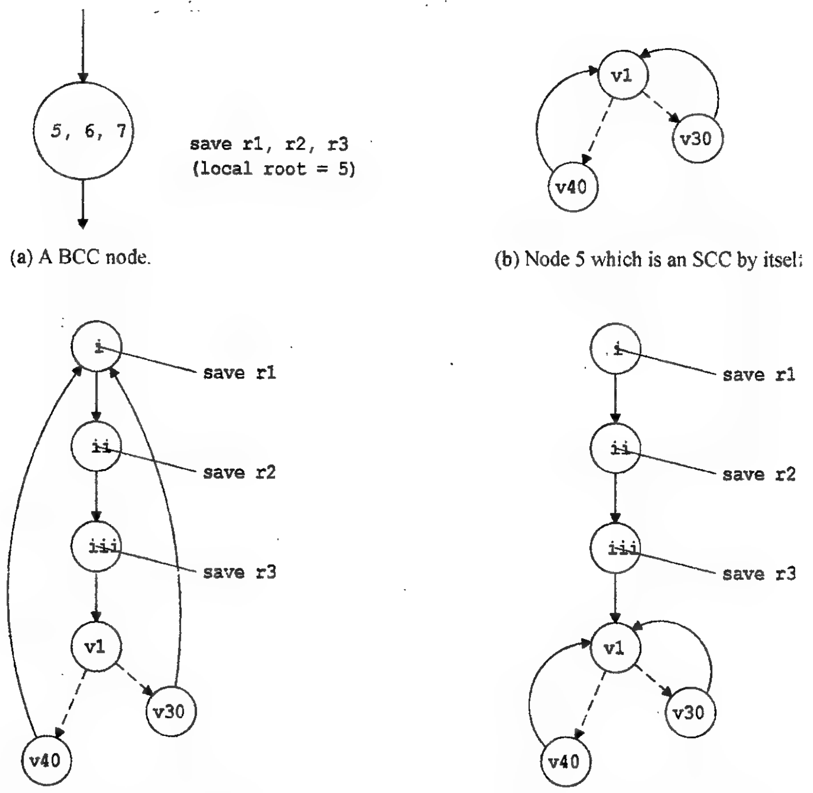
The last step is to generate register-saving operations. We start from the root node in the BCC flow graph moving down. At each node visited, we generate prolog for the registers belonging to its DKR except for the registers that are saved already. Fig. 8 shows prolog codes generated at each node. For example, at node 1, all registers in the corresponding DKR, r27 and r28, are saved. At node 2, the DKR contains r28 which is already saved

If we have to insert register-saving operations inside an SCC, we need an extra step as in Fig. 9. Fig. 9(a) shows a BCC node that includes node 5, 6, and 7. We assume that node 5 is by itself an SCC, and that it is the entry point of this BCC. Fig. 9(b) shows how node 5 looks like. When the algorithm decides that a prolog has to be generated at this BCC, the actual register-saving operations are generated on the entry node of it, which is node 5. Since node 5 is an SCC, the operations are generated on the starting basic block of this SCC, which currently includes only v1 as shown in Fig. 9(b). After inserting the register-saving operations, the flow graph becomes Fig. 9(c). Now the problem is clear: the register-saving operations are inside a loop. We need to adjust the targets of node v30 and v40, the children of v1, so that the prolog is hoisted out of the loop. The overall prolog generation algorithm is in Fig. 10.

5. Experiments

To measure the performance of our prolog tailoring algorithm, we took 8 procedures from xlisp 2.1, performed prolog tailoring on them, counted how many register-saving operations are generated, and finally computed the reduction rates compared to the numbers without prolog tailoring. Assuming all paths are selected equally, the average number of register-saving operations per path can be computed as follows.

$$AS = \sum_{i=1}^{NE} \frac{PE_i \cdot NS_i}{PT}$$



(a) A BCC node.

(b) Node 5 which is an SCC by itsel:

(c) Node 5 after register-saving operations inserted.

(d) Node 5 after adjusting back ed

Fig. 9. Generating register-saving operations inside an SCC node.

```
insert_regsave_code(node_t *n, regset_t tosave)
{
  generate register-saving operations for registers in "tosave" in the beginning of "n" ;
  if( "n" is an SCC byitself) {
    old_start ← the location after the generated operations;
    for( all branching operations in "n" )
      if( branching to "n" )
        adjust to branch to old_start;
  }
}

insert_prolog(bfgnode_t *n)
{
  if(there are registers in DKR(n) that are not saved yet)
  {
    v ← the registers in DKR(n) that are not saved yet;
    for( all local root nodes of "n", k )
      insert_regsave_code(k, v) ;
  }
  for( all children of "n", j)
    insert_prolog(j) ;
}
```

Fig. 10. Algorithm for generating register-saving instructions.

In above, PT is the number of executable paths; NE, the number of exit points; NS_i, the number of register-saving operations on a path ending with exit point *i*; PE_i, the number of possible paths reaching to exit point *I*; and finally, AS, the average number of register-saving operations.

The result in Table 2 shows that the average reduction rate is 17.4%. Excluding the most and the least reduction rates, it is 12.82%.

Table 2. The decreased number of register save instructions by prolog tailoring.

procedure	Before tailorig	After tailoring	difference	Reduction rate(%)
placeform	7	4.51	2.49	35.57
mark	8	5.50	2.50	31.25
sweep	9	6.50	2.50	27.78
xlpatprop	5	4.00	1.00	20.00
evlist	9	7.50	1.50	16.67
xlenter	6	5.79	0.21	3.50
evalh	9	8.70	0.30	3.33
cons	9	8.85	0.15	1.67
average				17.47
Normalized average				12.82

6 Conclusion

In this paper, we have proposed a prolog tailoring technique to reduce the overhead of prolog code in a procedure. Our algorithm generates register-saving operations as close as possible to the actual destruction points of the corresponding registers, but without unnecessary duplication and without incorrect modification of the original program's semantics. To achieve this, the proposed algorithm transforms the given control flow graph of a procedure into a BCC flow graph, compute DKRs on it, and generates prolog code. Through experimentations, we have observed that our method reduces the number of register-saving operations by 12.82% in average.

References

1. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, 1974
2. V. Aho, R. Sethi, and J. D. Ullman, *Compilers*, Addison Wesley, 1988
3. F. E. Allen, and J. Cocke, "A Catalogue of Optimizing Transformations," In *Design and Optimization of Compilers*, Prentice-Hall, 1972
4. K. Ebcioglu, R. D. Groves, K. C. Kim, G. M. Silberman, and I. Ziv, "VLIW Compilation Techniques in a Superscalar Environment," *ACM SIGPLAN Not.*, Vol.29, No.6, pp.36-48, 1994
5. J. Huang and D.J.Lilja, *Extending Value Reuse to Basic Blocks with Compiler Support*, IEEE Transactions on Computers, Vol. 49, No. 4, April 2000.
6. M. Lipasti, C. Wilkerson, and J. Shen, Value Locality and Load Value Prediction, Proc. Eighth Int'l Conf. Architecture Support for Programming Languages and Operating Systems, Oct. 1996.
7. M. Lipasti and J. Shen, Exceeding the Dataflow Limit via Value Prediction, Proc. 29th Ann. ACM/IEEE Int'l Symp. Microarchitecture, Dec. 1996.
8. K. O'Brien, B. Hay, J. Minish, H. Schaffer, B. Schloss, A. Shepherd, and M. Zaleski, "Advanced Compiler Technology for the RISC System/6000 Architecture," *IBM RISC System/6000 Technology*, SA23-2619, IBM Corporation, 1990
9. R. E. Tarjan, "Depth first search and linear graph algorithms," *SIAM J. Computing*, Vol.1, No.2, pp.146-160, 1972
10. G. Tyson and T. Austin, Improving the Accuracy and Performance of Memory Communication through Renaming, Proc. 30th Ann. Int'l Symp. Microarchitecture, Dec. 1997.

Constraint Programming

Hierarchical Constraint Satisfaction Based on Subdefinite Models

Dmitry Ushakov

Ledas Ltd.
Acad. Lavrentiev ave., 6
Novosibirsk, 630090, Russia
e-mail: ushakov@sib3.ru

Abstract. Hierarchical Constraint Satisfaction Problems (HCSPs) are at the centre of attention in the fields of computer aided design and user interfaces. Till recently, the algorithms proposed to solve the problems of this class focused only on its small subclasses (like problems with acyclic constraint graph or linear systems). Here we present a new family of hierarchical constraint satisfaction algorithms based on the mechanism of subdefinite models. The main advantage of the proposed algorithms is their applicability to a broad class of problems, including cyclic and non-linear ones.

1 Preliminaries

The mechanism of *subdefinite models* was proposed by the Russian scientist Alexander Narin'yani at the beginning of 1980s [1] independently on the western works in the field of constraint satisfaction problems (CSPs) [2]. Today we can say that this mechanism is a general-purpose apparatus to deal with CSPs. Using it, we have no restrictions on the domain of variables (taking into account both finite and continuous ones), the nature of constraints (dealing with both binary and n-ary constraints, implicit and explicit ones). The modern description of the mechanism of subdefinite models as well as the proof of its correctness can be found in [3]. We use the well-known notion of many-sorted algebraic models to feel ourselves freely in discussing general properties of the algorithms under consideration (and thus to apply our results to a broad class of problems, including finite, continuous and mixed ones).

With this chapter, we redefine the notion of HCSP (that was firstly proposed by Borning et al. [4]) in many-sorted terms.

1.1 Hierarchical Constraint Satisfaction Problem

As usually, let $\Sigma = (S, F, P)$ be a many-sorted signature (see [5] for details of this notion), X be an S-sorted set of variables, $\Sigma(X)$ be an extension of the signature Σ, where variables from X play the role of constant symbols, and $T_\Sigma(X)$ be the set of terms of the signature $\Sigma(X)$. (It is reasonable to suppose that Σ contains a predicate symbol of equality "$=$"$\in P_{ss}$ for each sort $s \in S$ with the following interpretation in any Σ-model M: $a =^M b \Leftrightarrow a = b$.) We define a $\Sigma(X)$-*constraint* c as an atom $p(t_1, \dots, p_n)$, where $p \in P_{s_1 \dots s_n}$, $t_i \in T_\Sigma(X)_{s_i}$ $(i = \overline{1,n})$. We denote the set of all variables occurring in constraint c by var(c). Let M be a Σ-model. We will say that a constraint $c \equiv p(t_1, \dots, t_n)$ *holds* in the model M iff there exists an estimate $\theta : X \to M$ of the variables X in the model M, s. t. $(\theta^*_{s_1}(t_1), \dots, \theta^*_{s_n}(t_n)) \in p^M$, where $\theta^* : T_\Sigma(X) \to M$ is the extension of the estimate θ to the set of $\Sigma(X)$-terms. In this case we will also say that *c holds on θ*.

A Hierarchical Constraint Satisfaction Problem (*HCSP*) over signature Σ is a pair (X, C), where X is a set of variables, and $C = \{C_0, C_1, \dots, C_m\}$ is a family of finite sets of $\Sigma(X)$-constraints. A constraint $c \in C_0$ is called *a required constraint*; constraints from $C_1 \cup \ldots \cup C_m$ are called *non-required* ones.

Let M be a Σ-model. For an HCSP $P = (X, C)$ we define the set $S^M_{P,0}$ of its *basic solutions* in the model M as follows:

$$S^M_{P,0} = \{\theta : X \to M | (\forall c \in C_0)\, c \text{ holds on } \theta\}.$$

The set of basic solutions thus is the set of all estimates, which satisfy the required constraints. For non-required constraints of level $i > 0$ we define the set $S^M_{P,i}$ as follows:

$$S^M_{P,i} = \{\theta : X \to M | (\forall j = \overline{0,i})(\forall c \in C_j)\, c \text{ holds on } \theta\}.$$

(Therefore, $S^M_{P,0} \supseteq S^M_{P,1} \supseteq \ldots \supseteq S^M_{P,m}$.) Real-world HCSPs are often over-constrained. It means that often not only $S^M_{P,m} = \emptyset$, but $S^M_{P,1}$ can be empty too. Therefore, we need a theory to choose what basic solution is better satisfied to non-required constraints. In other words, we need to compare basic solutions with respect to non-required constraints. A predicate $\text{better}^M_P \subseteq S^M_{P,0} \times S^M_{P,0}$ is called *a comparator* if it has the following properties for any $\theta, \phi, \psi \in S^M_{P,0}$:

1. *Irreflexivity*: $\neg\text{better}^M_P(\theta, \theta)$
2. *Transitivity*: $\text{better}^M_P(\theta, \phi) \wedge \text{better}^M_P(\phi, \psi) \Rightarrow \text{better}^M_P(\theta, \psi)$
3. *Correctness*: $(\forall i > 0)\, \theta \in S^M_{P,i} \wedge \phi \notin S^M_{P,i} \Rightarrow \text{better}^M_P(\theta, \phi)$

The set of solutions of an HCSP P in a model M w. r. t. to a comparator better^M_P is defined as follows:

$$S^M_{P,\text{better}^M_P} = \{\theta \in S^M_{P,0} | (\forall \phi \in S^M_{P,0})\, \neg\text{better}(\phi, \theta)\}.$$

1.2 Types of Comparators

We can classify some kinds of comparators. The simplest ones are so-called *predicate comparators*. Given a signature Σ, a Σ-HCSP $P = (X, C)$, a Σ-model M, and an estimate $\theta : X \to M$, define $H^M_{P,i}(\theta) \subseteq C_i$ as a set of constraints from C_i, which hold on θ. Using these sets we can build two comparators: lpb^M_P and gpb^M_P (these names are acronyms for *locally-predicate-better* and *globally-predicate-better*):

$$\text{lpb}^M_P(\theta, \phi) \Leftrightarrow (\exists k > 0) \begin{cases} (\forall i = \overline{1,k}) \;\; H^M_{P,i}(\theta) \supseteq H^M_{P,i}(\phi) \\ \qquad\qquad \wedge \;\; H^M_{P,k}(\theta) \supset H^M_{P,k}(\phi), \end{cases}$$

$$\text{gpb}^M_P(\theta, \phi) \Leftrightarrow (\exists k > 0) \begin{cases} (\forall i = \overline{1,k}) \;\; |H^M_{P,i}(\theta)| \geq |H^M_{P,i}(\phi)| \\ \qquad\qquad \wedge \;\; |H^M_{P,k}(\theta)| > |H^M_{P,k}(\phi)|. \end{cases}$$

The second group of comparators is metric ones. They are based on *an error function*. For an estimate θ and a $\Sigma(X)$-constraint c we define a non-negative real value $e^M(c, \theta)$, which is called the error of satisfaction of the constraint c on θ, and has the following property: $e^M(c, \theta) = 0 \Leftrightarrow c$ holds on θ. Given e^M, we define a comparator leb^M_{P,e^M} (an acronym for *locally-error-better*) as follows:

$$\text{leb}^M_{P,e^M}(\theta, \phi) \Leftrightarrow (\exists k > 0) \begin{cases} (\forall c \in C_1 \cup \ldots \cup C_k)\; e^M(c, \theta) \leq e^M(c, \phi) \\ \qquad\qquad \wedge\; (\exists c \in C_k)\; e^M(c, \theta) < e^M(c, \phi). \end{cases}$$

Comparators from geb (*globally-error-better*) family take into account global information about errors on each level. They can be expressed using a global error $g_{e^M}(C_i, \theta) = g(e^M, C_i, \theta)$, which summarizes all the errors of constraints from C_i on the estimate θ:

$$\text{geb}^M_{P,e^M,g}(\theta, \phi) \Leftrightarrow (\exists k > 0) \begin{cases} (\forall i = \overline{1,k}) \;\; g_{e^M}(C_i, \theta) \leq g_{e^M}(C_i, \phi) \\ \qquad\qquad \wedge \;\; g_{e^M}(C_k, \theta) < g_{e^M}(C_k, \phi). \end{cases}$$

We will use two global error functions: wc (an acronym for *worst-case*), and ls (*least-squares*):

$$\text{wc}(e^M, C_i, \theta) = \max_{c \in C_i} e^M(c, \theta),$$

$$\text{ls}(e^M, C_i, \theta) = \sum_{c \in C_i} (e^M(c, \theta))^2.$$

We will use notations wcb^M_{P,e^M} and lsb^M_{P,e^M} for geb-comparators based on wc and ls functions respectively.

Note, that predicate comparators can be unified with error ones by introducing a special error function pe, called *predicate error*:

$$\mathrm{pe}^M = \begin{cases} 0, & \text{if } c \text{ holds on } \theta, \\ 1, & \text{otherwise.} \end{cases}$$

Then lpb and gpb comparators can be expressed via leb and lsb ones respectively. Therefore, it is sufficient to consider only three comparators: leb, wcb, and lsb. However, one can propose simpler algorithms for hierarchical constraint satisfaction based on predicate comparators rather than on error ones.

2 Subdefinite Models

The algorithms of hierarchical constraint satisfaction, discussed below, are implemented in *subdefinite models* framework. Before considering the algorithms, we briefly remind the basic concepts of subdefinite models.

2.1 Subdefinite Extensions

In [3] we showed that subdefinite model approach allows one to find an approximation of the set of all solutions of a CSP (in our terminology, to find an approximation of the set of all basic solutions of an HCSP). This approximation is done by the means of achieving local subdefinite consistency. First, we build *subdefinite extensions* (*SD-extensions*) of universes of given Σ-model. If U is a universe, then its subdefinite extension, $^*U$ is a set of subsets of U, satisfying the following properties:

1. $\{\emptyset, U\} \subseteq {}^*U$.
2. $(\forall V, W \in {}^*U)\ V \cap W \in {}^*U$.
3. There are no infinite decreasing chains $(V \supset W \supset \ldots)$ in $^*U$.

Any subset V of U can be approximated in SD-extension $^*U$ as follows:

$$\mathrm{app}_{^*U}(V) = \bigcap_{V \subseteq W \in {}^*U} W. \qquad (1)$$

Let X be an S-sorted set of variables, M be a Σ-model, and $^*M$ be its SD-extension. *A subdefinite estimate* (*SD-estimate*) is an S-sorted mapping

$$\Theta = \{\Theta_s : X_s \to {}^*s^M | s \in S\},$$

which maps each $x \in X_s$ $(s \in S)$ into *a subdefinite value* $\Theta(x) \in {}^*s^M$. An SD-estimate Θ is *narrower* than another SD-estimate Φ iff $\Theta(x) \subseteq \Phi(x)$ for all $x \in X_s$, $s \in S$. An estimate θ is contained in an SD-estimate Θ (writing $\theta \in \Theta$) iff $\theta(x) \in \Theta(x)$ for all $x \in X_s$, $s \in S$. An SD-estimate, which does not contain any estimate, is called an empty SD-estimate (writing $\Theta = \emptyset$).

Given a signature Σ, and an S-sorted set of variables X, let c be a Σ-constraint, M be a Σ-model, and $^*M$ be its SD-extension. *A filtering* $\mathcal{F}_c$ of the constraint c is a mapping on the set of SD-estimates, satisfying the following conditions for any SD-estimates Θ, Φ:

1. *Contractness*: $\mathcal{F}_c(\Theta) \subseteq \Theta$.
2. *Monotonicity*: $\Theta \subseteq \Phi \Rightarrow \mathcal{F}_c(\Theta) \subseteq \mathcal{F}_c(\Phi)$.
3. *Correctness*: if c holds on θ, then $\theta \in \Theta \Rightarrow \theta \in \mathcal{F}_c(\Theta)$.
4. *Idempotency*: $\mathcal{F}_c(\mathcal{F}_c(\Theta)) = \mathcal{F}_c(\Theta)$.

An SD-estimate Θ is *consistent* w. r. t. a constraint c iff $\mathcal{F}_c(\Theta) = \Theta$. It is easy to see, that for any set C of constraints there exists unique SD-estimate Θ^*_C s. t.:

1. Θ^*_C is consistent w. r. t. each $c \in C$.
2. If an SD-estimate Φ is consistent w. r. t. each $c \in C$, then $\Phi \subseteq \Theta^*_C$.

Moreover, if there exists an estimate θ s. t. each $c \in C$ holds on θ, then $\theta \in \Theta^*_C$.

Fig. 1 shows the algorithm of finding the maximal consistent SD-estimate for a given set of constraints C. It uses a global structure `Ass`, where `Ass`(x) is a set of constraints from C where the variable x occurs.
The function returns `False` if the inconsistency is detected during filtering (it means $\Theta^*_C = \emptyset$). Choosing c in Q (the fourth line) can be arbitrary, but we use the principle "first in — first out", i. .e. regard Q as a queue.
In [3] we have proved that the call `Revise`(Θ^0, C), where $\Theta^0(x) = s^M$ for any $x \in X_s$, $s \in S$ produces Θ^*_C.

```
function Revise( in out $\Theta$, in $Q$ ) : boolean
begin
  while $Q \neq \emptyset$ do
    choose $c \in Q$;
    $\Phi \leftarrow \mathcal{F}_c(\Theta)$;
    for $x \in \mathrm{var}(c)$ do
      if $\Theta(x) \neq \Phi(x)$ then
        if $\Theta(x) = \emptyset$ then return False end if;
        $Q \leftarrow Q \cup \mathtt{Ass}(x)$
      end if
    end for;
    $Q \leftarrow Q \setminus \{c\}$;
    $\Theta \leftarrow \Phi$
  end while;
  return True;
end.
```

Fig. 1. The algorithm for achieving the maximal consistency

2.2 Solving an HCSP

We deal with a signature $\Sigma = (S, F, P)$, where there is a sort real $\in S$, and all constant, functional, and predicate symbols on it. We also deal with a Σ-model M, where real^M is the set of all real numbers, and all functional and predicate symbols have traditional interpretation ("+" as addition, "=" as equality, etc.) Suppose that all non-required constraints look like $x = \overline{0}$, where $x \in \mathrm{real}^M$, and $\overline{0} \in F_{\lambda,\mathrm{real}}$ with standard interpretation $\overline{0}^M$ as the real zero. (Below we consider the transformation of arbitrary HCSP to this form.) The need of globally processing all the constraints of the same level suggests us to deal with one constraint per level. Such a constraint has a form $\mathrm{zero}(x_1, \ldots, x_n)$ and is the reduction of a group of constraints $\{x_1 = \overline{0}, \ldots, x_n = \overline{0}\}$. It is reasonable to use the same schema of calculations for all the types of comparators. This schema is presented in fig. 2. The call `FilterNonRequiredConstraint` stands for the one of the procedures of filtering presented in fig. 3: `FilterLPB`, `FilterGPB`, `FilterLEB`, `FilterWCB`, or `FilterLSB`. Moreover, one can use different versions of comparators on each level of constraint hierarchy.

```
algorithm SolveHCSP( in $P = (X, \{C_0, \{c_1\}, \ldots, \{c_m\}\})$, out $\Theta$ )
begin
  [ build structure Ass ];
  for $s \in S$, $x \in X_s$ do
    $\Theta(x) \leftarrow s^M$   % assigning the maximal undefined values
  end for;
  if not Revise($\Theta, C_0$) then
    $\Theta \leftarrow \emptyset$
  else
    for $i = 1, \ldots, m$ do
      FilterNonRequiredConstraint($\Theta, c_i$)
    end for
  end if
end.
```

Fig. 2. The general algorithm for solving an HCSP

Procedure `FilterLPB` has nothing special, but others use an internal stack to implement the depth-first search of the solution. If the inconsistency is detected in some branch, the procedure returns to the previous suspended branch.

Procedure `FilterLEB` tries to narrow an SD-value of an argument variable as closer as possible to zero. However it has the following drawback. Suppose we have a non-required constraint $x = \overline{0}$ and required one

```
procedure FilterLPB( in out $\Theta$, in $c \equiv \text{zero}(x_1, \ldots, x_n)$ )
begin
  for $i = \overline{1, n}$ do
    if $0 \in \Theta(x_i)$ then
      $\Phi \leftarrow \Theta$;
      $\Phi(x_i) \leftarrow \text{app}_{{}^*\text{real}^M}(\{0\})$;
      if Revise($\Phi$, Ass($x_i$)) then $\Theta \leftarrow \Phi$ end if
    end if
  end for
end.
```

```
procedure FilterGPB( in out $\Theta$, in $c \equiv \text{zero}(x_1, \ldots, x_n)$ )
begin
  $\Theta^* \leftarrow \Theta$; $k^* \leftarrow 0$;
  push($\Theta, 1, 0$);
  while non-empty-stack do
    pop($\Theta, i, k$);
    if $i > n$ then if $k > k^*$ then $\Theta^* \leftarrow \Theta$; $k^* \leftarrow k$ end if
    else if $0 \in \Theta(x_i)$ then
      push($\Theta, i + 1, k$);
      $\Theta(x_i) \leftarrow \text{app}_{{}^*\text{real}^M}(\{0\})$;
      if Revise($\Theta$, Ass($x_i$)) then push($\Theta, i + 1, k + 1$) end if
    end if end if
  end while;
  $\Theta \leftarrow \Theta^*$
end.
```

```
procedure FilterLEB( in out $\Theta$, in $c \equiv \text{zero}(x_1, \ldots, x_n)$ )
begin
  $i \leftarrow 1$;
  while $i \leq n$ do
    push($\Theta, i$);
    $\Theta(x_i) \leftarrow \text{app}_{{}^*\text{real}^M}(\{\inf |\Theta(x_i)|\})$;
    if not Revise($\Theta$, Ass($x_i$)) then pop($\Theta, i$); end if
    $i \leftarrow i + 1$
  end while
end.
```

```
procedure FilterWCB/LSB( in out $\Theta$, in $c \equiv \text{zero}(x_1, \ldots, x_n)$ )
begin
  $w^* \leftarrow 0$;
  repeat
    $w \leftarrow g(\Theta)$;
    push($\Theta, \frac{w^*+w}{2}$);
    for $i = \overline{1, n}$ do $\Theta(x_i) \leftarrow \Theta(x_i) \cap [-\frac{w^*+w}{2}, \frac{w^*+w}{2}]$ end for
    if not Revise($\Theta, \cup_{i=\overline{1,n}}$Ass($x_i$)) then pop($\Theta, w^*$); end if
  until $w - w^* < \varepsilon$  % precision of calculation
end.
```

Fig. 3. The procedures of filtering better solution

$x = y - z$. (In fact, it means we have a non-required constraint $y = z$.) Suppose there is no basic solution θ with $\theta(x) = 0$. When we try to narrow an SD-value of x to zero, we need to assign x with some value $\mathrm{app}_{*\mathrm{real}^M}(\{a\})$, where $a > 0$. Roughly speaking, it means we try to filter a constraint $|y-z| = a$. This constraint has a disjunctive nature, since we do not know what constraint should be satisfied: $y = z + a$ or $z = y + a$. This lack of knowledge is often the reason of poor filtering.

The `FilterWCB` and `FilterLSB` procedures differ only in the function $g(\Theta)$:

$$g_{\mathrm{wcb}}(\Theta) = \max_{i=1}^{n} \sup \Theta(x_i),$$

$$g_{\mathrm{lsb}}(\Theta) = \sum_{i=1}^{n} (\sup \Theta(x_i))^2\}.$$

One can note that our algorithms are not complete in general sense: we cannot guarantee the existence of a solution in resulted subdefinite values. However, in real-world problems we can easily add weakest non-required constraints $x = a_x$ (with arbitrary chosen a_x) for *all* the variables of an HCSP under consideration, and thus the resulted values will be defined.

2.3 Transformation of an HCSP to a Simple Form

Remember, all the algorithms above deal with an HCSP, where all non-required constraints look like $x = \overline{0}$ for real variable x. How to transform any HCSP to this form? First, we need to extend our signature $\Sigma = (S, F, P)$ with a functional symbol diff_s for each sort $s \in S$: $\mathrm{diff}_s \in F_{ss,\mathrm{real}}$. This symbol should have the following interpretation in a Σ-model M: $\mathrm{diff}_s^M(a, b) = 0 \Leftrightarrow a = b$. For example, $\mathrm{diff}_{\mathrm{real}}$ can be implemented as $\mathrm{diff}_{\mathrm{real}}^M(a, b) = |a - b|$.

Consider a constraint $p(t_1, \ldots, t_n) \in C_k$ $(k > 0)$, where p is a predicate symbol, and $t_i \in T_\Sigma(X)_{s_i}$ $(i = \overline{1, n})$ are terms. Let $u_1, \ldots, u_n$ be new variables of sorts $s_1, \ldots, s_n$ respectively, and $v_1, \ldots, v_n$ be new variables of sort real. Then we transform our constraint into a set of ones:

- required constraint $p(u_1, \ldots, u_n)$,
- required constraints $\mathrm{diff}_{s_i}(u_i, t_i) = v_i$ for $i = \overline{1, n}$,
- non-required constraints $v_i = \overline{0}$ for $i = \overline{1, n}$.

2.4 Implementation Issues

These algorithms have been implemented in the framework of constraint programming environment *NeMo+* [6]. A set of benchmarks has been successfully solved. All these results are dropped here due to the space limitation, but will be presented in the full version of the paper.

3 Related Works

There are a number of algorithms for solving an HCSP. Most of them find a locally-predicate-better solution. Among others, the two most similar to our ones are *Indigo* [7] and *Projection* [8]. They are both intended for searching a locally-error-better solution and deal with interval values of variables. *Indigo* processes acyclic constraint graphs with numerical constraints and has the polynomial complexity. *Projection* processes systems of linear equations and inequalities but takes exponential time in the worst case. Of course, our general-purpose algorithm is not so efficient as these two, but it can be applied to non-linear systems with cyclic constraint graph: this is its main advantage.

Acknowledgements

The author is indebted to anonymous referees for their valuable comments.

References

1. Narin'yani, A.S.: Subdefiniteness and Basic Means of Knowledge Representation. Computers and Artificial Intelligence, Bratislawa **2(5)** (1983) 443–452
2. Mackworth, A.K.: Consistency in Networks of Relations. Art. Int. **8** (1977) 99–118
3. Ushakov, D.: Some Formal Aspects of Subdefinite Models. Preprint of A. P. Ershov Institute of Informatics Systems, Novosibirsk **49** (1998) (Also available via `http://www.iis.nsk.su/preprints/USHAKOV/PREPRINT/Preprint_eng.html`)
4. Borning, A., Freeman-Benson, B., Wilson, M.: Constraint Hierarchies. Lisp and Symbolic Computation **5** (1992) 223–270
5. Goguen, J.A., Meseguer, J.: Models and Equality for Logical Programming. LNCS **250** (1987) 1–22
6. Shvetsov, I., Telerman, V., Ushakov, D.: *NeMo+*: Object-Oriented Constraint Programming Environment Based on Subdefinite Models. LNCS **1330** (1997) 534–548
7. Borning, A., Anderson, R., Freeman-Benson, B.: Indigo: A local propagation algorithm for inequality constraints. Proc. 1996 ACM Symp. on User Interface Software and Technology (1996) 129–136
8. Harvey, W., Stuckey, P.J., Borning, A.: Compiling Constraint Solving Using Projection. LNCS **1330** (1997) 491–505

Using Constraint Solvers in CAD/CAM Systems

Vitaly Telerman

Dassault Systemes S.A. (France)
Institute of Informatics Systems, (Novosibirsk)
e-mail: vitali@wanadoo.fr

Abstract. We discuss different possibilities of using the Constraint Programming Solvers (CPS) in CAD/CAM systems. The *NeMo+* CPS, based on the approach of Subdefinite Models (SD-Models), and some its specializations are considered. The paper presents some components of a CAD/CAM system, where the *NeMo+* solver is used or can be used, discusses the advantages of this approach.

Introduction

Constraints are one of fundamental things that is intuitively known for the user in all areas of activity, including the CAD/CAM one [1]. Generally speaking, each interaction of two variables can be considered as a constraint. Constraints allow the user to specify the problem in the declarative manner. He doesn't need to specify "HOW to solve the problem", but only "WHAT a problem is necessary to solve".

Obviously, it is very important HOW constraints are solved. For solving the constraint satisfaction problem in the CAD/CAM system we propose to use the object-oriented solver *NeMo+* [2]. It is based on the well-known subdefinite models (SD-models) approach, proposed in the early 80th by Dr. A.S. Narin'yani [3].

We consider that the model of the designed entity (i.e. the designed product) in a CAD/CAM system consists of the *physical* part and the *functional* one.

The physical model is a decomposition of the product in assemblies, parts and design features. An assembly is a set of parts and or assemblies, the model can contain as many assembly levels as needed, a part is a set of features and a feature is a set of parameters that determine its properties and its behavior.

The functional model is a decomposition of the product in the different functions that it has to support. The product is divided into functions, the model can contain as many function levels as needed. Each elementary function (a function that can not be further decomposed) is implemented in the solver. A function can include features coming from different parts. The functional model allows the designer to work directly with functions, not necessarily knowing which parts are involved.

In this paper we propose an approach of a Constraint-Based CAD/CAM system, which, in our view, will have the following advantages:

- the designer has the possibility to work with the approximately known (or subdefinite) values of parameters (e.g. intervals — for real numbers, enumerations for discrete values, etc.);
- the solver returns to the designer both the validated subdefinite values, which can be more definite than the initial ones, and one of the exact solutions;
- the solver is able to solve together both the geometric constraints and the engineering ones. Thus it can considerably reduce the number of backtracks in the design process;
- the subdefinite model can be used all along the development of a design application since it can support the design knowledge acquisition, the implementation and the maintenance phases.

The paper is organized as follows. The section 1 gives a brief description of the constraint programming environment *NeMo+*. In the section 2 we present a *NeMo+* specialization for solving the geometric problems. The possibilities of using the constraint solver in conceptual and assembly design is discussed in the section 3. The use of *NeMo+* in Knowledge component of a CAD/CAM system is presented in the section 4. Section 5 gives a brief description of the use of *NeMo+* for solving time-based problems in digital manufacturing and product data management. The last section contains the conclusions and further plans.

1 NeMo+ Constraint Programming Environment

The object-oriented constraint programming environment *NeMo+* is a state-of-the-art constraint solver that, besides the traditional constraint satisfaction algorithm, incorporates a number of constraint programming techniques: root locating, symbolic transformations and differentiation, heuristics for partial satisfaction problems, specialized module for solving geometric constraints.

The standard *NeMo+* environment has the following peculiarities:

1. An extended set of predefined data types which may have finite as well as infinite domains of values. It includes an extensive library of basic (i.e. implemented in C++) constraints for such data types as set, Boolean, integer and real, strings, and any other types defined by the user. The domain of a variable can be represented by a single value, by enumeration of possible values, by interval or multi-interval values. The choosing of data type representations allows the user to the compromise between the solution quality and the calculation time.

2. Availability of high-level facilities for specification of problem-oriented constraints and data types. It includes a high-level language for specification of systems of constraints. This language is a purely declarative one and allows the user to describe a system of constraints as a collection of formulas. Object-oriented properties of the *NeMo+* language are used to define the structure of a model and to define problem-oriented data types and constraints from the base ones. In addition, the language includes sophisticated means for controlling the constraint propagation process.

3. Solving the direct and the inverse problems on the same specification of the problem. Taking into account the initial values or parameters, the solver defines itself what problem should be solved (direct, inverse, or both).

4. Use of the method of subdefinite models to satisfy the system of constraints. The main feature of the method of SD-models is that it uses a single algorithm of constraint propagation to process data of different types. This allows one to solve mixed of constraints including simultaneously set, Boolean, integer and real variables and constraints on them. Moreover, *NeMo+* proposes different techniques to find an exact solution, including the optimal one.

5. *NeMo+* can process constraints defined by tables, including database ones. It chooses all reliable data from the table/database and uses them in another constraints as subdefinite data. It should be mentioned that tables/databases themselves may contain the subdefinite values.

The algorithm of computations implemented in SD-models is a highly parallel data-driven process. Modification of the values of some variables in the common memory automatically results in calling and executing those constraints for which these variables are arguments. The process halts when the execution of constraints does not change the variables values.

During the computation of one model, the solver starts twice. At first, it checks the consistency of the model and improves the initial subdefinite values. Then, it starts for finding one of the exact solutions. Both results, the subdefinite (consistent) values and the exact solution are persistent in the CAD/CAM system.

The *NeMo+* solver has been implemented jointly by Russian Research Institute of Artificial Intelligence (Moscow-Novosibirsk) and by Institute of Informatics Systems (Novosibirsk).

Summarizing all these properties we can say that *NeMo+* can be used in different parts of CAD/CAM systems such as:

- Sketcher (geometric solver);
- Conceptual and Assembly design;
- Knowledge based engineering;
- Digital manufacturing and Product Data Management.

Moreover, *NeMo+* can be placed in the kernel of a CAD/CAM system (so-called feature platform) to provide a general-purpose solution for both update engine and user interaction.

2 Constraint Solver for Geometric Modeler

Geometric applications in CAD/CAM area all have a fundamental requirement to maximize the productivity of the designer by enabling the efficient construction and modification of geometric models.

In our view, each geometric problem belongs to the class of constraint satisfaction problems, i.e. its specification is a declarative one, which contains a set of objects connected by a set of geometric constraints.

Present CAD systems are merely based on parametric or variational design. The best known of geometric solvers is DCM (Dimensional Constraint Manager by D-Cubed Ltd.) [4]. DCM is based on an algorithm for computing the solution for a subset of all possible equations of geometry and dimensions, using purely algebraic methods. The usual arithmetic operations are used including square roots. DCM considers in detail the equations that are obtained when points, lines and circles on a plane are defined by means of relative distance and angle constraints. The main result is that these equations can be solved algebraically for a significant class of configurations.

In [5] the DCM method is seen as a propagational solver; solving constraints that can sequentially be constructed on a drawing board, using ruler and compass. According to [5] propagational solvers offer robustness, accuracy and speed. However, they are restricted to relatively simple problems. The main problem is that mathematical constraints which determine other product characteristics than those related to geometry alone also have to be taken into account [6]. These constraints cannot easily be solved in existing CAD systems as they are highly coupled and non-linear. The second problem is the problem of measurement accuracy, and tolerancing. It is almost evident that, due to the measuring instrument's accuracy, some geometric values like lengths, distances and angles are approximately known in real-life problems. Different techniques can be used to solve such kind of problems. DCM is able to take into account the approximately known values of dimensions via inequalities, but it always returns only one exact solution.

The main advantage of *NeMo+* is that it is able to solve geometric problems with exact and/or interval values of parameters jointly with non-geometric (so-called, engineering) constraints, and it returns two kind of results: the exact solution, and the subdefinite values.

For solving the geometric problems in the standard *NeMo+* it is necessary either to specify all significant constraints (the theorem of cosine, the formulae of Heron, the sum of angles in the triangle, etc.) or to specify the problem in terms of high-level objects like triangles, rectangles, trapeze, etc, in which the coherence relationships are included yet. Obviously, both solutions are not acceptable, when *NeMo+* is used as a solver in Sketcher programs. It is more preferable that the solver make itself the decision what relationships are necessary to be taken into consideration for solving the given problem. In order to do that, a specialized library was implemented in the *NeMo+* environment, which can be considered as a *NeMo+* geometric modeler. In our view, the *NeMo+* geometric modeler is an "intelligent" instance solver, which is able to solve well-constrained, under-constrained, and over-constrained (but consistent) problems. Using the partial constraint satisfaction tecniques it also can solve the over-constrained (and inconsistent) geometric problems. The *NeMo+* geometric modeler was implemented by E.V. Roukoleev. The partial constraint satisfaction algorithms were implemented by D.M. Ushakov.

The geometric modeler allows *NeMo+* to compute the model, which contains only the elementary geometric objects (points, lines, angles, ...) and constraints (perpendicularity, parallelism, distance, ...). Using this information and the intermediary results, obtained during the constraint propagation, the modeler generates new constraints on parameters of the model. The modeler uses three methods for changing the model: unification, decomposition, and synthesis.

Unification. The basic geometric objects like points, lines and planes are considered for this method, and for each of them the concept of "index" with the following properties is determined (for objects of the same type):

1. Index(a)=Index(b)<=>a=b<=>Distance(a,b)=0; where Index: $G \rightarrow Z$.
2. If $x = F(a_1, a_2, \dots, a_N)$, $y = F(b_1, b_2, \dots, b_N)$, and

$$Index_F(Index(a_1), \dots, Index(a_N)) = Index_F(Index(b_1), \dots, Index(b_N)),$$

then $x == y$, where $Index_F : Z \times Z \times \dots \times Z \rightarrow Z$. Thus, if in the model there are constraints of equivalence or of equality to zero of distances, the unification of the appropriate objects is done. And if the arguments of functions are unified, their results are unified too.

Decomposition. Usually the complex relations are expressed through the more simple ones. During the interpretation of the complex relation the modeler try to determine if some of its more simple components exist in the model or not. If a more simple relation exists, then the modeler doesn't create a new component and uses the existing one.

Synthesis. When we have a lot of constraints linked to the same object, sometimes it is possible to create for this object a stronger relation. For example,
`On(PointA,LineAB)&On(PointB,LineAB)->LineAB=On(PointA,PointB);` In the case of three distances AB, BC and AC, this technique allows the modeler to find out the contradictions before setting up the coordinate values:
$AB + BC >= AC$; $AB + AC >= BC$; $AC + BC >= AB$.
Using this technique it is possible to solve not only the common geometric problems but also the optimization ones, containing engineering constraints. For example, one can find such a configuration of a complex sketch than the sum of areas (engineering constraints) of some closed contours consists exactly 30 percents of the area of the whole sketch.

The first step of validation of the modeler has been achieved with success for a set of examples in 2D-geometry.

3 Constraint Solver in Conceptual and Assembly Design

A CAD/CAM products mostly consist of a number of parts (which are made of features) that are connected to each other. The ideal product modeling system should therefore be able to support the modeling of all parts and their connections. Assembly constraints provides information on which component is connected to which other component in what way (face-contact), thus representing a model of an assembly. A CAD/CAM system must maintain the consistency of the designed product.

The design of a product can be thought of as a top-down (Conceptual Design) and/or bottom-up (Assembly Design) processes. Both of them can be considered as a sequence of phases, each of which adds information to the product model.

In the early phases of conceptual design, in which all global product requirements are gathered into the model, the designer does not yet want to think about all kinds of details that are not directly related to these requirements. In these phases, the designer only wants to specify those parts and constraints that are needed to satisfy the global requirements.

An assembly is a collection of parts, assembly features (coordinate systems, datum entities) other assemblies, and assembly properties. In the assembly the designer takes the ready-to-use parts and connects them by constraints according to the product requirements. If changes occur in one component the constraints can take care of the propagation of these constraints. This propagation can be authomatically done by solvers like *NeMo+*. Moreover, in the case when there exists libraries, catalogues of standard elements (parts, products), *NeMo+* can be used for the intelligent search of such elements. The *NeMo+* object-oriented language allows the designer to specify the query in high-level terms of the given data domain. It is possible to associate to the query more sofisticated requirements such as systems of equations, inequalities, rules (conditions), diagrammes, indicate the possible alternatives, etc. The end-user query is associated to the *NeMo+* model, elaborated and implemented by an expert. The solver, as we have mentioned before, returns the subdefinite result (the set of possible solutions) and one exact solution.

For example, a fragment of an expert model, which provides the choice of a bearing type from the given catalog, looks as follows:

```
B:Bearing;
B.P0 == B.Fr + (0.5 *B.Fa);  B.L10=(B.C / B.P)^B.p;
if B.Types2 == 2 then
 B.d == [ 3.0, 160.0 ];
 B.D == [ 10.0, 240.0 ];
 if ((311 <== B.Num)/\(B.Num <== 486)) // Kind of joints
 then ((-30 <== B.T)/\(B.T <== 110));  // Limits of temperature
 end;
 if (((B.T>>20) /\ (B.T<<120) /\ (B.nu>>5.01) /\ (B.nu<<400))
   \/((B.T>>20) /\ (B.T<<120) /\ (B.nu1>>5.0) /\ (B.nu1<<400)))
 then DIAGRAMME2 ( B.nu, B.T, B.nu1 );
 end;
end;
```

It should be noted once more that the values of parameters in all assembly or conceptual design components can be subdefinite. They become more and more exactly in the design process, when new components and new constraints arise in the product. This is the main difference between the proposed approach and the well-known existing industrial CAD/CAM systems, which assume that components are completely specified before assembly modelling is performed.

4 Constraint Solver in Knowledge Component

The use of the Knowledge component in a CAD system gives to the designer the following possibilities:

- Create and manage rules and knowledge bases;
- Check rules and knowledge base compliancy after design;
- Invoke knowledge base advisor during design.

The *NeMo+* environment can be used in the Knowledge component as a basic solver, which provides the rules checking, tables computation, optimization, constraint satisfaction, and solving a complex system of equations, inequalities, including real numbers, integers, strings, booleans, sets, and user-defined types.

Currently *NeMo+* is incorporated in the Knowledge component of a CAD/CAM system, where it represents a very clear concept for the user: a set of equations and inequalities. Such a set is defined in terms of mathematical equalities and inequalities and can include arithmetic, trigonometric and other standard mathematical functions. The user can arbitrary divide the parameters included in the set of equations (e.g. cost, material, distance, angle, area, volume, etc.) into two groups: inputs and outputs. Values of the input parameters are taken from the product, the output ones should be calculated by the solver. So, the interactive changing of input values enforces outputs to be recalculated by *NeMo+*. This behavior allows the user to optimize any parameter under design by easy switch between inputs and outputs. The implementation of the set of equations has been done by D.M.Ushakov.

5 Constraint Solver in Manufacturing & Data Management

Increasingly, CAD/CAM research is concerned with developing an integrated approach, incorporating the activities of design, manufacturing, process management and maintenance.

An advanced CAD/CAM system will have the following capabilities:

1) Integrate design-to-order and scheduling so as to calculate delivery dates.

2) Allocate resources and schedule the work of different teams, throughout the product life cycle, from design, through production and to disposal.

Obviously, the advantages of *NeMo+* for solving a calendar planning and job-shop scheduling problem is the possibility to deal with intervals of beginning, ending, and duration of jobs [7]. In order to solve more efficiently time scheduling problems, a specialized library of *NeMo+*, a JobShopScheduler, was implemented.

JobShopScheduler is a solver for use by applications with a need for solving job-shop scheduling problems such as well-known bridge building planning problem. This solver deals with jobs (that may, in turn, consist of smaller subjobs), which need to be scheduled according to constraints that link those jobs together. The constraints may concern jobs' precedence, their possibilities to perform at a given time, simultaneously with another jobs etc. The important feature of the solver is the presence of the notions of a resource, resource pool, and resource allocation. This allows to state and solve complex optimization problems where jobs' processing requires certain resources and resources have limited capacity.

JobShopScheduler is implemented as a C++ library, which includes a set of basic constraint types derived from *NeMo+* ones and, also, high-level constraints expressing most often used relationships between jobs and resources. This library was implemented by V.S.Markin.

6 Conclusion

In the paper, we proposed the way to use constraint programming solvers in different components of a CAD system. In order to allow end-users to make a maximum profit, it is necessary that features support approximately known values of designed entities, and these values should be persistent in the model. The *NeMo+* constraint programming environment (or the solver with the same capabilities) is proposed to be used.

NeMo+ is implemented in C++ under *Windows* and UNIX platforms. There are no restrictions on the kind of constraints it can solve. One can build *NeMo+* specialized solvers in order to make it more efficient.

The applicability of this approach was proven by prototyping the geometric, and JobShopScheduler solvers in a CAD/CAM programming development environment, and by an integration of the *NeMo+* solver in the Knowledge component of the CAD/CAM system.

The forthcoming work will include the extention of *NeMo+* possibilities to process not only the functions implemented in its libraries, but also the external functions. Another interesting topic for *NeMo+* is the distributed and collaborative design. We hope, that in the future, *NeMo+* (or a *NeMo+*-like sover) will be integrated in the kernel of a CAD/CAM system for providing a general-purpose solution for the update engine of a CAD/CAM system.

References

[1] Montanari, U. *Networks of Constraints: Fundamental Properties and Application to Picture Processing*, Information Science,

[2] Shvetsov I., Telerman V., Ushakov D. *NeMo+: Object-Oriented Constraint Programming Environment Based on Subdefinite Models*, LNCS **1330** (1997).

[3] Narin'yani, A.S. *Subdefiniteness and Basic Means of Knowledge Representation*, Computers and Artificial Intelligence, Bratislawa, **2**, No.5, (1983), 443 – 452.

[4] *The 2D DCM Manual (Version 3.7.0)*, D-Cubed Ltd., July, 1999, 314p.

[5] Brown Associates Inc. *Applicon's GCE: a strong technical framework*, June 1993.

[6] Thornton A.C. *Constraint specification and satisfaction in embodiment design*, PhD thesis, University of Cambridge, 1993.

[7] Narin'yani A.S., Borde S.B., Ivanov D.A. *Subdefinite Mathematics and Novel Scheduling Technology*, Artificial Intelligence in Engineenring, **11**, (1997).

A Graphical Interface for Solver Cooperations

Laurent Granvilliers and Eric Monfroy

IRIN, University of Nantes
F-44322 Nantes cedex 3, France
e-mail: {granvilliers,monfroy}@irin.univ-nantes.fr

Abstract. We propose a language composed of basic graphical components. By assembling these components as in a Lego game, solver cooperations can be visualized. The advantage is to represent by simple figures complex cooperations that usually requires tedious descriptions. We illustrate our language by implementing some well-known cooperative solvers.

1 Introduction

Solver cooperation is now well-known as a concept for improving efficiency and performance of constraint solvers. Generic solvers are generally far too inefficient for solving numerous real-life problems. However, a large part of these problems can often be handled by "incomplete" but specific and efficient solvers; and a solver can pre-process constraints in order to ease and speed-up a second solver.

The most usual type of cooperations (that we call *ad-hoc* cooperations) are based on one cooperation concept (*e.g.*, sequential solving process, or concurrent communication) and one solving strategy: the solvers are known, and the rooting of constraints through the solvers is fixed *a priori*. Examples of such cooperations are [3, 6, 8]. Implementing *ad-hoc* cooperations is a tedious task that involves several different problems, such as implementing communication between solvers, fixing interoperability problems, filtering constraints, synchronizing solving processes, and in the worst case re-implementing solvers from scratch.

On the one hand, cooperation languages [7, 5] recently emerged as a new concept for designing and automatically implementing (such as in [7]) solver cooperations as expressions of a calculus-like language. However, cooperation expressions quickly become difficult to read. Moreover, interactions and communications between solvers are not explicit, but are hidden in the definitions of the primitives for building expressions.

On the other hand, the concept of coordinating a number of activities, running concurrently in a parallel and distributed fashion, has recently received wide attention (*e.g.*, see [9]). Visual interfaces to such languages already exist, such as Visifold [4] for the control-driven coordination language Manifold [1].

In this paper, we propose a language for graphically designing solver cooperations. This language is composed of some few basic components from which more complex bricks and solver cooperations are built such as in a Lego game. Usual patterns of cooperation (such as sequential, concurrent, and parallel solving processes), and standard control on constraint routing (such as conditional, fixed-point, selections) can easily be designed linking solver, control, filter, and selection agents with channels of communication. Complex cooperations are then built connecting these patterns of constraint processing. This language aims at representing graphically in a unified and simple way solver cooperations. The growing capacity of this language is tremendous: first, patterns of cooperations can become new bricks of the language, and second, new basic components can easily be integrated. Moreover, adding a new component correspond to implementing a new module that does not interact with previous pieces of code.

We illustrate the practicality of our language by visualizing some *ad-hoc* cooperations (such as the ones of [3, 2]) that normally require long descriptions.

The outline of this paper is the following: definitions for constraints, solvers, filters are presented in Section 2. Basic graphical components are described in Section 3 in terms of communicating agents. Using these components, some standard patterns of cooperation are designed in Section 4, before visualizing some well-known ad-hoc cooperations in Section 5. We finally conclude in Section 6.

2 Framework

Let $\mathcal{D}$ be a set called the universe, $\mathcal{F}$ a set of function symbols, $\mathcal{R}$ a set of relation symbols, $\Sigma = (\mathcal{D}, \mathcal{F}, \mathcal{R})$ a structure, and $\mathcal{X} = \{x_1, \ldots, x_n\}$ a set of variables. A *constraint language* $\mathcal{L}$ is a non-empty set of first order

$(\Sigma, \mathcal{X})$- formulae. Given a constraint c on variables $x_1, \ldots, x_n$, let ρ_c denote the underlying relation on $\mathcal{D}^n$. The relation ρ_c associated to the constraint c on $x_{i_1}, \ldots, x_{i_l}$ is extended to the set $\rho_c^+ = \{(v_1, \ldots, v_n) \in \mathcal{D}^n \mid (v_{i_1}, \ldots, v_{i_l}) \in \rho_c\}$. A *constraint store* C is given as a set $\{c_1, \ldots, c_m\}$ of constraints from $\mathcal{L}$ interpreted as the conjunction $c_1 \wedge \cdots \wedge c_m$. The solutions of C (denoted by $Sol(C)$) are defined as:

$$Sol(C) = \bigcap_{i=1}^{m} \rho_{c_i}^+$$

$\mathcal{L}_S$ represents the set of stores built upon the constraint language $\mathcal{L}$. We can now define the notion of solver in our scheme.

Definition 1 (Solver). *Consider a constraint language $\mathcal{L}$. Then, a solver S on $\mathcal{L}$ is a computable function $S : \mathcal{L}_S \longrightarrow \mathcal{L}_S$.*
A solver is said:

$$\begin{array}{l} \textit{correct:}\ \ \textit{if}\ \forall C \in \mathcal{L}_S,\ Sol(S(C)) \subseteq Sol(C) \\ \textit{complete:}\ \textit{if}\ \forall C \in \mathcal{L}_S,\ Sol(C) \subseteq Sol(S(C)) \end{array}$$

With respect to Definition 1, Gröbner basis computation, Simplex, Gaussian elimination, factorization of polynomials, trigonometric transformations are solvers. No property is required *a priori* for solvers. However, some properties of solver cooperations are induced from solver properties (see *e.g.*, dispatcher in Section 4).

We say that some solvers $S_1, \ldots, S_k$ on $\mathcal{L}_1, \ldots, \mathcal{L}_k$ can cooperate on a constraint language $\mathcal{L}$ if for all $i \in [1, k]$, $\mathcal{L}_i \subseteq \mathcal{L}$. Stores from $\mathcal{L}_i$ are called *admissible constraints* of S_i on $\mathcal{L}$.

The role of filters is very important in a cooperation on a language $\mathcal{L}$. They select parts of constraints stores, *i.e.*, subsets of stores in order to: 1) select constraint stores a solver S on $\mathcal{L}' \subseteq \mathcal{L}$ can actually handle, *i.e.*, the admissible constraints of S on $\mathcal{L}$, and 2) treat efficiently subsets of stores by specific solvers.

Definition 2 (Filter). *Consider a constraint language $\mathcal{L}$. A filter on $\mathcal{L}$ is a computable function $\varphi : \mathcal{L}_S \longrightarrow \mathcal{L}_S$ such that:*

$$\forall C \in \mathcal{L}_S,\ \varphi(C) \subseteq C$$

Usual filters are $\varphi_{\text{eq\_poly}}$ to filter polynomial equations, φ_{lin} to filter linear constraints, *etc.*

Property 1. A filter is a complete and non correct solver.

Filters can be combined using intersection, union, and complementary operators to compose more complex filters. The results are the expected standard set operators on stores of constraints. Consider two filters φ_1 and φ_2 on $\mathcal{L}$. Then, for all $C \in \mathcal{L}_S$, we have:

$$\begin{array}{l} (\varphi_1 \cup \varphi_2)(C) = \varphi_1(C) \cup \varphi_2(C) \\ (\varphi_1 \cap \varphi_2)(C) = \varphi_1(C) \cap \varphi_2(C) \\ \bar{\varphi}(C) = C \setminus \varphi(C) \end{array}$$

3 Basic Graphical Components

We design a set of graphical —basic— components that can be combined to implement solver cooperations. The underlying model is based on *agents* —solvers, filters, selectors, *etc.*— acting on constraints. An agent receives data on input *ports*, transforms them, and puts the resulting data on output ports. Ports of agents are connected by *channels*, where a channel transfers a constraint store from an output port of an agent to an input of another agent.

The basic graphical components are presented in Figure 1. Their precise meanings will be explained in the following.

3.1 Communication

Communications of constraint stores are modeled by ports —holes on agents— and channels —linkers of ports. A *port* either models an *input* of an agent, or an *output*. A *channel* implements a one-to-one (from an output port to an input port) communication of constraint stores.

In the following, we propose agents achieving solver computations, and filtering, controling, and redirecting communications.

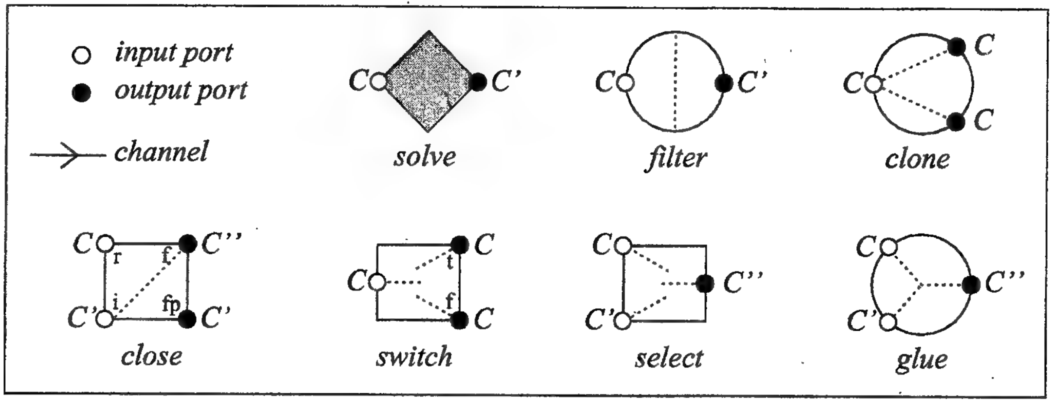


Fig. 1. Basic graphical components.

3.2 Solving Agents

Solving agents (Primitive *solve* in Figure 1) capture the computational part of cooperations. In fact, most of existing systems integrate a restricted set of algorithms (such as Gröbner bases, consistency techniques, interval methods, *etc.*) that are seen as black-box solvers. Consider a solver S. Then, from an input constraint store C (available on the input port), if C belongs to the constraint language of S, then S is applied on C ($S(C) = C'$), and the new store C' is delivered on the output port; otherwise, S is not applied, and $C' = C$.

3.3 Transformer Agents

Transformer agents are essentially solvers processing constraints by means of set operations: restriction, union, conjunction, *etc.* We briefly discuss four kinds of useful operations.

- A *filter agent* (Primitive *filter* in Figure 1) applies a filter φ (see Section 2) on C to extract a subset of the constraints verifying some property, *i.e.*, $C' = \varphi(C)$ such that $C' \subseteq C$. A filter is generally used to preprocess a constraint store before applying a specific solver.
- *Cloning* (Primitive *clone* in Figure 1) a constraint store C consists in duplicating C on every output ports. Note that the combination of several cloning agents increases the number of clones: $n-1$ combined cloning agents lead to n clones. Cloning agents are useful for realizing different usual tasks of cooperation, such as modeling concurrent algorithms, and processing of sub-stores.
- *Gluing* (Primitive *glue* in Figure 1) constraint stores C and C' means generating their union $C'' = C \cup C'$. This operation is performed when all input stores are available on input ports. Typically, it gathers together results generated by cooperative solvers acting on the same constraint store (competitive concurrent solvers), or acting on disjoint sub-stores (cooperative concurrent solvers).
- The *selection* (Primitive *select* in Figure 1) of two input constraint stores C and C' transfers just one of them to the output port. We have either $C'' = C$ or $C'' = C'$.

3.4 Control Agents

A control agent manages the rooting of constraint stores during the solving process. We identify agents modeling switches and fixed-point computations.

- A *switch* (Primitive *switch* in Figure 1) is based on a P function from stores to Booleans: P checks whether input constraint stores verify a given property or not. Consider a P function and an input store C: if $P(C)$ is true, then C is transfered to the output port t, otherwise to the output port f. Switches represents conditional of [7]: they are very important to dynamically control solver cooperations.
- A more complex kind of switch (Primitive *close* in Figure 1) is devoted to *fixed-point* computations. It is based on the detection of equivalent consecutive input constraint stores. For this purpose, such an agent has a memory —a constraint can be stored between consecutive applications—, two input ports i (input) and r (re-enter), and two output ports f (follow) and fp (fixed-point). The computation processes as follows:

- Initially, the memory is empty. First time a constraint store C is received on i (input), it is stored in the memory, and put on port f (follow).
- Then, when a constraint store C' arrives on r (re-enter), it is compared with the memory. If they are equivalent, C' is put on port fp (fixed-point) and the memory is cleared and reset for next use —the fixed-point is just detected.
- If they are different, C' is put on port f (follow) and the memory is updated with C'.

4 Standard Patterns of Cooperation

We now describe two patterns involved in numerous cooperations.

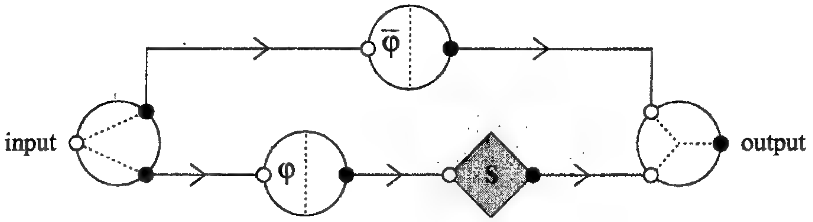


Fig. 2. Solver protection.

When using a *solver protection* (Figure 2) solvers process only subsets of their admissible constraints (*i.e.*, constraints they can effectively handle), while the rest of the input store is preserved, and used to create the new constraint store. More precisely, the input store is first cloned. Then,

– on one branch, the store is filtered by φ to extract from a constraint store C the admissible constraints of S. S is then applied on the resulting store;

– on the other branch, the filter $\bar{\varphi}$ (*i.e.*, the complementary of φ) filters $C \setminus \varphi(C)$.

The output of S, and the non-admissible constraints of S are then glued together: the final output of the protection is: $(C \setminus \varphi(C)) \cup S(\varphi(C))$.

Property 2. Consider a solver S, and φ to filter its admissible constraints. If S is complete, then the protection of S is also complete.

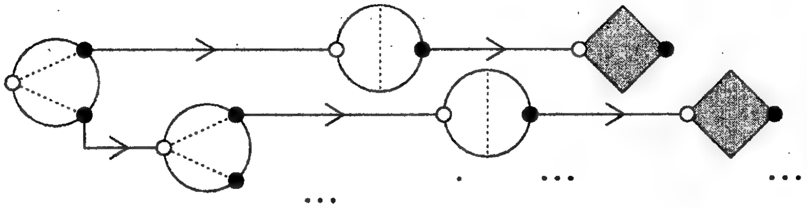


Fig. 3. Dispatcher of a store to solvers.

A *dispatcher* (Figure 3) distributes constraint store to n solvers (each of them being associated with a specific filter) using $n-1$ cloning agents. Note that it can also be combined with a solver protection to preserve the whole of the input store.

5 Modeling Existing Systems

We design three existing cooperative solvers to illustrate the feasibility of our approach.

The system presented in [3] is devoted to linear constraints: domains of variables are reduced with an interval solver while a Simplex-like solver tries to detect inconsistency of stores and to fix variables. As soon as new

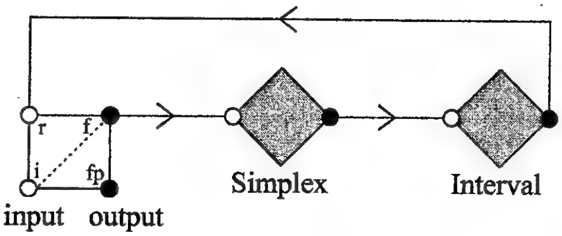


Fig. 4. Cooperation Simplex-interval solver on linear constraints.

information is deduced by one of the solvers, it is communicated to the other one. The process terminates when a fixed-point is reached, *i.e.*, none of the solver is able to deduce new facts anymore. This cooperation is visualized in Figure 4. The solvers are applied in sequence, each one processing the whole constraint store. The detection of a fixed-point is realized by a closure agent.

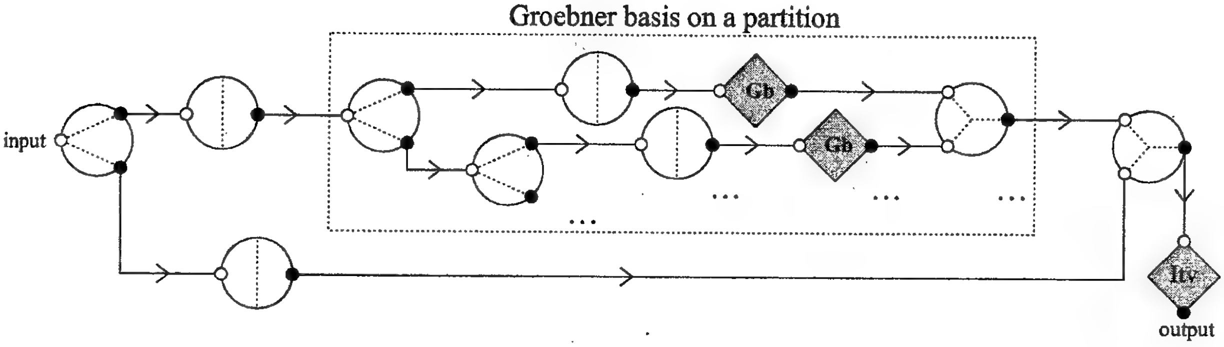


Fig. 5. Cooperation Gröbner basis-interval solver on arbitrary constraints.

The cooperation of Gröbner basis computation used as a preprocessing for an interval solver is visualized in Figure 5. The input constraint store is filtered in order to extract the set of polynomial equations. A set of Gröbner bases is then computed for a partition of the set of polynomial equations (construction cloning-filter-solver). The input of the interval solver is the union of the computed Gröbner bases and the input constraints that are not polynomial equations.

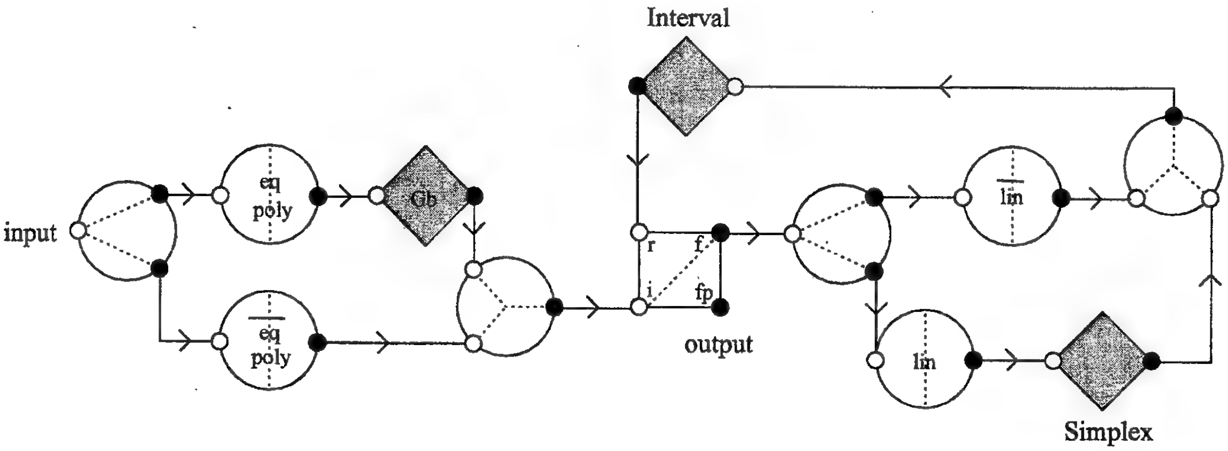


Fig. 6. Cooperation Gröbner basis-Simplex-interval solver.

Figure 6 illustrates the following cooperation: a Gröbner basis is generated for polynomial equations, and they are combined with the other input constraints. Then, a fixed-point of the Simplex and an interval solver applied in sequence is computed. A filter of linear constraints is necessary in front of the Simplex, while the interval solver handles all constraints (linear and nonlinear constraints are combined after the application of Simplex).

6 Conclusion

We have proposed here a language for visualizing solver cooperations in a unified and graphical manner. This language is composed of basic components that are then linked together by channels in order to graphically represent solver cooperations. Complex cooperations that usually require long explanations are described by a simple figure in our language. The growing capacity of the language are tremendous since integrating a new basic component corresponds to adding a module in a component based-framework and does not provoke any side-effect.

We are confident in the practical realization of our language to automatically implement solver cooperations from their graphical descriptions: cooperation features are similar to primitives of **BALI** (which has already been implemented), and more complex visual interface (such as [4]) have already been realized for complete coordination languages (such as Manifold [1] which requires several complex communication and interaction features).

In the future, we plan to extend our language by introducing several types of communication in order 1) to enable solvers waiting for complementary constraints, and 2) to manage disjunctions of constraints (*e.g.*, several possible solutions) as different constraint stores requiring different solving processes.

References

1. F. Arbab. *Manifold2.0 reference manual.* CWI, Amsterdam, The Netherlands, May 1997.
2. F. Benhamou and L. Granvilliers. Automatic Generation of Numerical Redundancies for Non-Linear Constraint Solving. *Reliable Computing*, 3(3):335–344, 1997.
3. H. Beringer and B. De Backer. Combinatorial problem solving in constraint logic programming with cooperative solvers. In *Logic programming: formal methods and practical applications.* Elsevier Science Publisher B.V., 1995.
4. P. Bouvry and F. Arbab. Visifold: A visual environment for a coordination language. In Paolo Ciancarini and Chris Hankin, editors, *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*, pages 403–406. Springer-Verlag, April 1996.
5. C. Castro and E. Monfroy. A Control Language for Designing Constraint Solvers. In *Proceedings of Andrei Ershov Third International Conference Perspective of System Informatics, PSI'99*, volume 1755 of *Lecture Notes in Computer Science*, pages 402–415, Novosibirsk, Akademgorodok, Russia, 2000. Springer-Verlag.
6. P. Marti and M. Rueher. A Distributed Cooperating Constraints Solving System. *International Journal on Artificial Intelligence Tools*, 4(1&2):93–113, 1995.
7. E. Monfroy. The Constraint Solver Collaboration Language of BALI. In D.M. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems 2*, volume 7 of *Studies in Logic and Computation*, pages 211–230. Research Studies Press/Wiley, 2000.
8. E. Monfroy, M. Rusinowitch, and R. Schott. Implementing Non-Linear Constraints with Cooperative Solvers. In K. M. George, J. H. Carroll, D. Oppenheim, and J. Hightower, editors, *Proceedings of ACM Symposium on Applied Computing (SAC'96), Philadelphia, PA, USA*, pages 63–72. ACM Press, February 1996.
9. G. A. Papadopoulos and F. Arbab. Coordination models and languages. *Advances in Computers*. 46: The Engineering of Large Systems, 1998.

Program Analysis

Abstract Computability of Non-Deterministic Programs over Various Data Structures

Nikolaj S. Nikitchenko

Department of the Theory of Programming
Faculty of Cybernetics, Kiev National University
Vladimirskaja 64, Kiev, 01033, Ukraine
e-mail: nikit@unislav.kiev.ua

Abstract. Partial multi-valued functions represent semantics of non-deterministic programs. The notion of naturally computable partial multi-valued function is introduced and algebraic representations of complete classes of naturally computable functions over various data structures are constructed.

1 Introduction

The title of this paper is a clear reminiscence of A.P. Ershov's articles "Abstract computability on algebraic structures" [1] and "Computability in various domains and bases" [2]. This tight connection of titles is not accidental and has a long history. The story goes back into 1984 when the author being on a sabbatian leave from the Kiev State University spent half a year in Novosibirsk at the A.P. Ershov's department of the Computer Center of the Siberian Branch of the Soviet Academy of Sciences. During his stay in Novosibirsk the author had the possibility to study the intentions of Ershov's works on computability, was fascinated by his ideas and tried to follow them in his own investigations. The author is grateful to A.P. Ershov for support in his work on the topic.

The main objective of Ershov's works on computability was the necessity to develop for computer sciences their own fundamental conceptions of the computability theory [1]. Such a theory must define computability for various subject domains and different systems of basic operations; clearly distinguish combinatorial and "executable" aspects of computability; be independent of specific program syntax and mechanisms of program evaluation [2].

The author's research on computability were based on the following ideas of A.P. Ershov:
- the notion of abstract computability must be oriented on abstract models of programs,
- abstract computability has a relative character,
- the notion of determinant[1] can be used for the definition of function computability.

To realise these ideas we
- construct abstract but powerful models of programs,
- define exact definitions of computability, which satisfy the described requirements,
- study the properties of introduced notions of computability.

Such definitions were first developed for the compositional model of programs [3, 4]. The notions of natural and determinant computability were introduced and the complete classes of functions and compositions over different classes of named data were described. In this paper we extend this approach to new more general classes of program models, based on composition nominative principles [5]. The main extensions concern computability over nominative data for non-deterministic programs.

[1] A.P. Ershov understood determinants as sets of special terms constructed over given algebraic system [2].

The paper is structured in the following way:

– first, we define composition nominative systems, which can be considered as abstract powerful program models,

– then, we discuss questions of computability in programming languages and define the notion of natural computability of partial multi-valued functions, which represent semantics of non-deterministic programs,

– at last, complete classes of computable partial multi-valued functions over different specializations of nominative data structures are described.

2 Composition Nominative Approach to Program Definition

The main goal of the approach is to construct a clear hierarchy of adequate models of program of various levels of abstraction and generality. Dialectical logic developed by G.W.F. Hegel and his followers is used as a gnoseological (epistemological) foundation of this approach.

The approach is based on the following principles, which specify the main program notions.

Development principle (rising from abstract to concrete): the notion of program should be introduced as a process of its development, which starts from abstract understanding capturing essential program properties and proceeds to more and more concrete considerations thus gradually revealing the notion of program in its richness.

Applicativity (functionality) principle: at the highest abstraction level programs can be considered as functions which being applied to input data can produce output data.

Function nominativity principle: programs can be presented as names denoting functions which being applied to input data can produce output data.

Compositionality principle (V. Red'ko [6]): programs can be considered as functions which map input data into output data, and which are constructed from simpler programs (functions) with the help of special operations, called compositions.

Descriptivity principle: programs can be considered as descriptions (complex names) which denote functions constructed from simpler functions with the help of compositions.

These principles introduce five notions: data, function, function name, composition and description, which form the pentad of main program notions. Formalizations of such notions are usually based on the notion of set, thus giving set-theoretic formalizations of programs. Still, there are proposals to use instead the notion of function [7]. Here we will follow this way considering a function-theoretic approach.

Principle of function-theoretic formalization: program notions are formalized on a base of a function-theoretic approach.

Please note that we do not reduce the notion of set to the notion of function, but treat these notions as mutually dependent on each other.

Functions, which maps elements of A into R, are considered in the most general way as partial multi-valued functions. In this case functions are not uniquely represented by their graphs, therefore we will additionally take into consideration the sets of elements on which functions can be undefined (undefinedness sets). For example, function $f = [1 \mapsto 1, 1 \mapsto 2, 1 \mapsto, 2 \mapsto, 3 \mapsto 1]$ has a binary relation $\{(1,1),\ (1,2),\ (3,1)\}$ as its graph and a set $\{1,2\}$ as its undefinedness set. We will use the following notation for the classes of functions:

– $D \overset{m}{\to} D$ – partial multi-valued functions,

– $D \to D$ – partial single-valued functions,

– $D \overset{t}{\to} D$ – total single-valued functions.

Program models on high abstraction levels can be presented as composition nominative systems [5]. Such a system may be considered as a triple of the following simpler systems: composition, nominative, and denotational systems. Composition system defines semantic aspects of programs, nominative system defines program descriptions (syntactic aspects), and denotational system specifies meanings of descriptions. Here we will consider only composition systems which are triples of the form $< D, F, C >$, where D is a set of data, on which programs are defined, $F \subseteq (D \overset{m}{\to} D)$ is a class of partial multi-valued functions, representing program semantics, and C is a class of compositions over F, representing program construction means.

These definitions are specialised for more concrete levels. We can distinguish three main levels: abstract, Boolean, and nominative levels [5]. The last level is the most interesting level for programming. On this level program data are considered as nominative data, which are constructed hierarchically with the help of naming relations.

For given sets of names V and basic elements W the class of nominative data $ND(V, W)$ can be presented by the following recursive definition:

$$ND(V,W) = W \cup (V \xrightarrow{m} ND(V,W)).$$

Main operations over nominative data with the name v as a parameter are the following.

- Naming operation $\Rightarrow v$. Being applied to some data d it yields the nominative data having the only component with the name v and the value d.
- Partial multi-valued denaming operation $v\Rightarrow$. Being applied to some nominative data d it yields one (arbitrary chosen) value of the name v, if at least one component with the name v is in d.
- Deleting operation $\backslash v$. Being applied to some nominative data d it deletes one (arbitrary chosen) component with the name v, if such a component is in d.
- Definiteness operation $v!$. Being applied to some data d it yields the empty nominative data $\emptyset$, if v has at least one value in d, and d, if v has no value in d.

We also use non-deterministic *choice* operation, which on d yields d or $\emptyset$.

Concretizations of nominative data can represent various data structures, such as records, arrays, sets, tables, etc. [4,5]. For example, a set $\{s_1, s_2, ..., s_n\}$ can be presented as a nominative data $[1 \mapsto s_1, 1 \mapsto s_2, ..., 1 \mapsto s_n]$, where 1 is treated as a standard name. Thus, we can formulate the following principle.

Data nominativity principle: program data structures can be presented as concretizations of nominative data.

Having defined composition nominative systems as powerful program models (models of programming languages), we can now specify a special computability for such models.

3 Computability in Programming Languages

Traditional programming languages are usually called universal languages. It means that their programs define computable functions, and vice versa, any computable function may be represented by a certain program, written in such a language. But more thorough investigation reveals a number of difficulties, which are concerned with our understanding of computability in programming languages. Usually computability is understood as computability of n-ary functions defined on integers or strings. Such computability may be called Turing computability. But programming languages also work with other data structures and it turns out that for these structures programming languages, which are considered as universal, may not be universal. That is: their programs cannot represent all computable functions definable on these structures [5].

So, the computational completeness of programming languages is not a trivial problem and calls for specific further investigations.

The completeness problem is not the only aim of our investigation. Now it is a common opinion that programs should be developed successively from abstract specifications via more concrete representations up to detailed implementations in chosen programming languages. And it is very important to connect completeness and computability problems with stages of program development. We intend to introduce such unified notions of computability and completeness that can be applied to every stage of program development and can be easily transformed when moving from stage to stage of development. Such a kind of computability should be applicable to data structures of different abstraction levels and is called abstract computability [1]. In fact, such computability is a relative computability – relative to data structures and operations over them.

Another facet of the problem is formulation of simple and clear descriptions of complete classes of computable functions on each level of abstraction. We shall construct algebraic representations of such classes. It means, that a complete class will be described as the closure of some basic functions under a certain (and very simple) class of compositions (operators over functions).

Many results are currently available in this area. We only mention [1,8–14]. However, despite the richness of the available results, the attempt to apply them to our problems runs into various difficulties.

The point is that many approaches postulate certain requirements which, first, are far from obvious and themselves require substantiation (e.g., the existence of a universal computable function, as assumed by Strong, Freedman, Moschovakis) and second, are often inapplicable to specific data structures (e.g., the requirement of data enumerability of Mal'tsev's enumeration approach, or the requirement of Goedelisation of Wagner's approach (see refs. in [2]). Recall that we are concerned with computability defined in terms of data structures of programming languages. We will therefore try to motivate the proposed formalization of computability by using weaker postulates, from which other postulates may be obtained as corollaries (as suggested by Gandy [8] and Scott [9]). In other words, we will attempt to identify the key ideas of abstract computability, which can be combined to obtain concrete results.

We will study computational completeness of the classes of functions over nominative data. The difficulty of the problem lies in the fact that the notion "computability over complex data structures" by itself must first be defined and then, only on this basis, complete classes of functions can be described.

Here we restrict our consideration only by computability over finite data structures, which is called natural computability.

4 Natural Computability of Partial Multi-Valued Functions

In order to formalise computability of functions over finite data structures, we first need to define such data. This is a difficult question, and we will accordingly adopt the following strategy: we will first define a special form of finite data structure and subsequently reduce data of other forms to this special form.

Our intuitive notion of a finite data structure is the following: any such datum d consists of several basic (atomic) components $b_1, ..., b_m$, organised (connected) in a certain way. If there are enumerably many different forms of organisation for finite data structure, each of these data can be represented in the (possibly non-unique) form $(k, < b_1, ..., b_m >)$, where k is the **datum code** and the sequence $< b_1, ..., b_m >$ is the **datum base.** Data of this form are called **natural data** [3]. More precisely, if B is any set and Nat is the set of natural numbers, then the set of natural data over B is the set $Nat(B) = Nat \times B^*$.

A set D is called a set of **finite data structure** (over B with respect to nat), if a set B and a total multi-valued injective[2] mapping $nat : D \overset{m}{\to} Nat(B)$ are given. This mapping nat is called the **naturalization** mapping, and the partial single-valued inverse mapping $nat^{-1} : Nat(B) \to D$, denoted by $denat$, is called the **denaturalization** mapping.

Very often the denaturalization mapping is called the abstraction mapping. We prefer to start with naturalization mapping as primary, because our definitions are developed in the direction from abstract levels to concrete ones.

The introduction of natural data and naturalization mappings enables us to reduce computability over D to special computability over $Nat(B)$, which is called code computability. To define this type of computability we should recall that in natural data the code collects all known information about datum components. Thus, code computability should be independent of any specific processing tools of the elements of the set B and can use only those tools which are independent of B and are explicitly exposed in natural data. The only explicit information in natural data is the datum code and the length of the datum base. Therefore in code computability the datum code plays a major role, while the elements of the datum base are "extras" that virtually do not affect the computations. The elements of a datum base may be only used to form the base of the resulting datum. These considerations lead to the following definition.

A function $g : Nat(B) \overset{m}{\to} Nat(B)$ is called **code computable** if there exists a partial recursive multi-valued function $h : Nat^2 \overset{m}{\to} Nat \times Nat^*$ such that for any $k, m \in Nat$, $b_1, ..., b_m \in B$, $m \geq 0$

$$g(k, < b_1, ..., b_m >) = (k', < b_{i_1}, ..., b_{i_l} >),$$

if and only if

$$h(k, m) = (k', < i_1, ..., i_l >), 1 \leq i_1 \leq m, ..., 1 \leq i_l \leq m, m \geq 0.$$

In other words, in order to compute g on $(k, < b_1, ..., b_m >)$, we have to compute h on (k, m), generate a certain value $(k', < i_1, ..., i_l >)$, and then try to form the value of the function g by selecting the components of the sequence $< b_1, ..., b_m >$ pointed to by the indexes $i_1, ..., i_l$.

We are ready now to give the main definition of this section.

A function $f : D \overset{m}{\to} D$ is called **naturally computable** (with respect to given B and nat) if there is a code computable function $g : Nat(B) \overset{m}{\to} Nat(B)$ such that $f = denat \circ g \circ nat$.

We may consider natural computability as a generalization (relativization) of enumeration computability. In fact, for $B = \emptyset$ code computability reduces to partial recursive computability on Nat, and natural computability reduces to enumeration computability (wrt nat). Natural computability may be also used to define computability of polymorphic functions. Therefore, the notions of code and natural computability defined above are quite rich.

Having defined the notion of natural computability we can now construct algebraic representations of complete classes of naturally computable partial multi-valued functions for various data structures which are considered as specializations of nominative data.

[2] A multi-valued function is injective, if it yields different values on different arguments.

5 Complete Classes of Computable Partial Multi-Valued Functions

In this extended abstract we present without proofs only a few results describing complete classes of computable functions over simple subclasses of nominative data. Appropriate naturalization mappings, inducing properties of these data structures, can be easily defined.

In order to describe such complete classes we will use the following compositions (exact definitions see in [5]): multiplication (functional composition) $\circ$, iteration (loop) $*$, overlaying (overriding) ∇.

5.1 Computability over Named Data

A class of named data is a special subclass of nominative data with single-valued naming and is defined by the following recursive equation:

$$NAD(V,W) = W \cup (V \xrightarrow{n} NAD(V,W)),$$

where $A \xrightarrow{n} R$ is a set of finite single-valued mappings.

The class of computable functions over $NAD(V,W)$ depends on the abstraction level, on which V and W are considered. Here we present only two cases determined by finite and countable sets of names.

Let $V = \{v_0, ..., v_m\}$ be a finite set, W be an abstract set. Then data of the set $NAD(V,W)$ are called V-finite W-abstract named data.

Theorem 1. *The complete class of naturally computable partial multi-valued functions over the set of V-finite W-abstract named data precisely coincides with the class of functions obtained by closure of the set of functions* $\{\Rightarrow v_0, ..., \Rightarrow v_m, v_0\Rightarrow, ..., v_m\Rightarrow, v_0!, ..., v_m!, choice\}$ *under the set of compositions* $\{\circ, *, \nabla\}$.

Let $V = \{v_0, v_1, ...\}$ be an enumerable set, W be an abstract set ($V \cap W = \emptyset$). Since V is enumerable, any name from V can be recognised and generated. Therefore, elements of the set V will be used not only as names but also as basic values. In other words, we will consider the set of named data $NAD(V, W \cup V)$. Such data are called V-enumerable W-abstract named data.

In order to describe complete classes over such data, we should use the following additional functions.

- Functions over V: successor $succ_V$, predecessor $pred_V$ and constant $\bar{v}_0$.
- Equalities: $=v_0, =\emptyset$.
- Unary predicates: $\in V, \in W$.
- Binary functions: *as* (naming), *cn* (denaming), *ext* (removal) and predicate *ec* (existence of named component), such that for $v \in V$, $d \in D$ $as(v,d) = \ \Rightarrow v(d)$, $cn(v,d) = \ v\Rightarrow(d)$, $ex(v,d) = \backslash v(d)$, $ec(v,d) = v!(d)$.

Theorem 2. *The complete class of naturally computable partial multi-valued functions over the set of V-enumerable W-abstract named data precisely coincides with the class of functions obtained by closure of the set of functions* $\{\Rightarrow v_0, \Rightarrow v_1, \in V, \in W, \bar{v}_0, =v_0, =\emptyset, succ_V, pred_V, as, cn, ex, ec, choice\}$ *under the set of compositions* $\{\circ, *, \nabla\}$.

5.2 Computability over Nominative Data

In comparison with named data, nominative data allow multi-valued naming. To work efficiently with such data we have to consider a more specific abstraction level introducing equality on W. Nominative data of this level will be called W-equational data. To present computable functions over such data we additionally introduce a subtraction function $\backslash$ of nominative data and binary union composition $\sqcup$ defined by the formula $(f \sqcup g)(d) = f(d) \cup g(d)$. In this case the equality on W is derivable. For the class of V-finite W-equational nominative data we can formulate the following result.

Theorem 3. *The complete class of naturally computable partial multi-valued functions over the set of V-finite W-equational nominative data precisely coincides with the class of functions obtained by closure of the set of functions* $\{\Rightarrow v_0, ..., \Rightarrow v_m, v_0\Rightarrow, ..., v_m\Rightarrow, v_0!, ..., v_m!, choice, \backslash\}$ *under the set of compositions* $\{\circ, *, \sqcup\}$.

5.3 Computability over Sequences

Traditional data structures may be considered as concretizations of nominative data. Here we present the completeness result for functions over sequences.

Let B be an abstract set and $Seq(B)$ be the set of all sequences, hierarchically constructed from elements of B. The set $Seq(B)$ may be defined by recursive definition $Seq(B) = B \cup Seq(B)^*$.

The structure $Seq(B)$ has been investigated in different works. We shall use the notations of [15]. Four new functions are introduced: *first*, *tail*, *apndl*, *is-atom*. Also, we need a composition, called construction: $[f,g](d) =< f(d), g(d) >$.

Theorem 4. *The complete class of naturally computable partial multi-valued functions over the set $Seq(B)$ precisely coincides with the class of functions obtained by closure of the set of functions $\{first, tail, apndl, is\text{-}atom, choice\}$ under the set of compositions $\{\circ, *, [\,]\}$.*

In a similar way we can also generalize the completeness results for compositional databases presented in [16].

6 Conclusion

In this short paper we defined the notion of natural computability for partial multi-valued functions, which represent semantics of non-deterministic programs. This computability is a special abstract computability, that satisfy the main requirements formulated by A.P. Ershov. The complete classes of naturally computable functions are described for simple cases of nominative data. The proposed technique can be used for more rich data structures. The notion of natural computability forms a base for the notion of determinant computability of compositions. The obtained results can be used to study computational completeness of programming, specification and database query languages of various abstraction levels.

References

1. A.P. Ershov. Abstract computability on algebraic structures.– In: A.P. Ershov, D. Knuth (Eds) Algorithms in modern mathematics and computer science. Berlin: Springer (1981) 397–420
2. A.P. Ershov. Computability in arbitrary domains and bases, Semiotics and Informatics, No. 19 (1982) 3–58. In Russian.
3. N.S. Nikitchenko. On the construction of classes of generalized computable functions and functionals, UkrNIINTI, techn. report No 856 Uk-84, Kiev (1984) 51 p. In Russian.
4. I.A. Basarab, N.S. Nikitchenko, V.N. Red'ko. Composition databases, Kiev, Lybid' (1992) 192 p. In Russian.
5. N. Nikitchenko. A Composition Nominative Approach to Program Semantics. Technical Report IT-TR: 1998-020. Technical University of Denmark (1998) 103 p.
6. V.N. Red'ko. Composition of programs and composition programming. Programmirovanie, No 5 (1978) 3–24. In Russian.
7. K. Grue. Map theory. Theoretical Computer Science, v. 102(1) (1992) 1–133
8. R. Gandy. Church's thesis and principles for mechanisms. The Kleene Symp. Eds. J. Barwise, et. al, Amsterdam: North-Holland (1980) 123–148
9. D. Scott. Domains for denotational semantics. LNCS, v. 140 (1982) 577–613
10. Y.N. Moschovakis. Abstract recursion as a foundation for the theory of algorithms. Lect. Notes Math, v. 1104 (1984) 289-362
11. A.J. Kfoury, P. Urzyczyn. Necessary and sufficient conditions for the universality of programming formalism. Acta Informatica, v. 22 (1985) 347–377
12. E. Dahlhaus, J. Makowsky. The Choice of programming primitives for SETL-like programming languages. LNCS, v. 210 (1986) 160–172
13. J.V. Tucker, J.I. Zucker. Deterministic and nondeterministic computation, and Horn programs, on abstract data types, J. Logic Programming, v. 13 (1992) 23–55
14. V.Yu. Sazonov. Hereditarily-finite sets, data bases and polynomial computability. Theoretical Computer Science, v. 119 (1993) 187–214
15. J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Communs. ACM, v. 21 (1978) 613-641
16. I.A. Basarab, B.V. Gubsky, N.S. Nikitchenko, V.N. Red'ko. Composition models of databases. Extending Inf. Syst. Technology, II Int. East-West Database Workshop, Sept. 25-28, 1994, Klagenfurt, Austria (1994) 155-163

On Lexicographic Termination Ordering with Space Bound Certifications

G. Bonfante, J.-Y. Marion, and J.-Y. Moyen

Loria, Calligramme project
B.P. 239, 54506 Vandœuvre-lès-Nancy Cedex, France
e-mail: {Guillaume.Bonfante,Jean-Yves.Marion,Jean-Yves.Moyen}@loria.fr

Abstract. We propose a method to analyse the program space complexity, based on termination orderings. This method can be implemented to certify the runspace of programs. We demonstrate that the class of functions computed by first order functional programs over free algebras which terminate by Lexicographic Path Ordering and admit a polynomial quasi-interpretation, is exactly the class of functions computable in polynomial space.

1 Introduction

Motivations. There are several motivations to develop automatic program complexity analysis :

1. The control of the resources consumed by programs is a necessity, in software development.
2. There is a growing interest in program resource certifications. For example, Benzinger [2] has implemented a prototype to certify the time complexity of programs extracted from Nuprl [6]. Various systems have been defined to control resources in functional languages, see Weirich and Crary [7] and Hofmann [12].
3. Our approach is based on well-known termination orderings, used in term rewriting systems, which are easily implemented.
4. The study of program complexity is of a different nature compared to program verification or termination. It is not enough to know what is computed, we have to know how it is performed. So, it gives rise to interesting questions whose answers might belong to a theory of feasible algorithms which is not yet well established.

Our results. We consider first order functional programs over any kind of constructors, which terminate by Lexical Path Ordering (LPO). We demonstrate that the class of functions which are computed by LPO-programs admitting polynomially bounded quasi-interpretations, is exactly the class of functions which are computed in polynomial space. (See Section 3.2 for the exact statement.) This resource analysis can be automatized. Indeed, Nieuwenhuis in [21] has proved that termination by LPO is NP-complete. To find a quasi-interpretation is not too difficult in general, because the program denotation turns out to be a good candidate.

Complexity and termination orderings. Termination orderings give rise to interesting theoretical questions concerning the classes of functions for which they provide termination proofs. Weiermann [23] has shown that LPO characterizes the multiple recursive functions and Hofbauer [10] has shown that Multiset Path Ordering (MPO) gives rise to a characterization of primitive recursive functions. While both of these contain functions which are highly unfeasible, the fact remains that many feasible algorithms can be successfully treated using one or both. Quasi-interpretations allows us to tame the complexity of treated algorithms. Indeed, it has been established [20] that functions computed by programs terminating by MPO and admitting a polynomial quasi-interpretation are exactly the polynomial time computable functions. This last result might be compared with the one of Hofbauer. Analogously, the result presented in this paper might be compared with Weiermann's one.

Others related characterizations of poly-space There are several characterizations of PSPACE in Finite Model Theory, and we refer to Immerman's book [13] for a complete presentation. A priori, Finite Model Theory approach is not relevant from the point of view of programming languages because computational domains are infinite. But recently, Jones [14] has showed that polynomial space languages are characterized by mean of read-only functional programs. He has observed a closed relationship with a characterization of Goerdt [9].

On infinite computational domains, characterizations of complexity classes go back to Cobham's seminal work [5]. The set of polynomial space computable functions has been identified by Thompson [22]. Those characterizations are based on bounded recursions. Hence, they do not directly study algorithms and so they are not relevant to automatic complexity analysis.

Lastly, from the works of Bellantoni and Cook [1] and of Leivant [17], purely syntactic characterizations of polynomial space computable functions were obtained in [18, 19]. The underlying principle is the concept of ramified recursion. From the point of view of automatic complexity analysis, the main interest of this approach is that we have syntactic criteria to determine the complexity of a program. However, a lot of algorithms are ruled out. Several solutions have been proposed to enlarge the class of algorithms captured. Hofmann [11] has proposed a type system with modalities to deal with non-size increasing functions, e.g. the functions *max* and *min*. Another solution is to introduce a ramified version of termination orderings MPO [19], which delineates polynomial time and polynomial space computable functions.

2 First Order Functional Programming

Throughout the following discussion, we consider three disjoint sets $\mathcal{X}, \mathcal{F}, \mathcal{C}$ of variables, function symbols and constructors.

2.1 Syntax of Programs

Definition 1. *The sets of terms, patterns and function rules are defined in the following way:*

$$\begin{array}{lll}
\textit{(Constructor terms)} & \mathcal{T}(\mathcal{C}) \ni u & ::= \mathbf{c} \mid \mathbf{c}(u_1, \cdots, u_n) \\
\textit{(Ground terms)} & \mathcal{T}(\mathcal{C},\mathcal{F}) \ni s & ::= \mathbf{c} \mid \mathbf{c}(s_1, \cdots, s_n) \mid \mathtt{f}(s_1, \cdots, s_n) \\
\textit{(terms)} & \mathcal{T}(\mathcal{C},\mathcal{F},\mathcal{X}) \ni t & ::= \mathbf{c} \mid x \mid \mathbf{c}(t_1, \cdots, t_n) \mid \mathtt{f}(t_1, \cdots, t_n) \\
\textit{(patterns)} & \mathcal{P} \ni p & ::= \mathbf{c} \mid x \mid \mathbf{c}(p_1, \cdots, p_n) \\
\textit{(rules)} & \mathcal{D} \ni d & ::= \mathtt{f}(p_1, \cdots, p_n) \to t
\end{array}$$

where $x \in \mathcal{X}$, $\mathtt{f} \in \mathcal{F}$, and $\mathbf{c} \in \mathcal{C}$[1]*. The size $|t|$ of a term t is the number of symbols in t.*

Definition 2. *A program $\mathcal{E}(\mathtt{main})$ is a set $\mathcal{E}$ of $\mathcal{D}$-rules such that for each rule $\mathtt{f}(p_1, \cdots, p_n) \to t$ of $\mathcal{E}$, each variable in t appears also in some pattern p_i. All along the paper, we assume that the set of rules is implicit and we just write* `main` *to denote the program.*

Example 3. *The following program is intended to compute the Ackermann's function. Take the set of constructors to be $\mathcal{C} = \{\mathbf{0}^0, \mathbf{S}^1\}$. We shall note as exponents the arity of the symbols.*

$$\begin{array}{c}
\mathtt{Ack}(\mathbf{0}, n) \to \mathbf{S}(n) \\
\mathtt{Ack}(\mathbf{S}(m), \mathbf{0}) \to \mathtt{Ack}(m, \mathbf{S}(\mathbf{0})) \\
\mathtt{Ack}(\mathbf{S}(m), \mathbf{S}(n)) \to \mathtt{Ack}(m, \mathtt{Ack}(\mathbf{S}(m), n))
\end{array}$$

2.2 Semantics

The signature $\mathcal{C} \cup \mathcal{F}$ and the set $\mathcal{E}$ of rules induce a rewrite system which brings us the operational semantics. We recall briefly some vocabulary of rewriting theories. For further details, one might consult Dershowitz and Jouannaud's survey [8] from which we take the notations. The rewriting relation $\to$ induced by a program `main` is defined as follows $t \to s$ if s is obtained from t by applying one of the rules of $\mathcal{E}$. The relation $\xrightarrow{*}$ is the reflexive-transitive closure of $\to$. Lastly, $t \xrightarrow{!} s$ means that $t \xrightarrow{*} s$ and s is in normal form, *i.e.* no other rule may be applied. A ground (resp. constructor) substitution is a substitution from $\mathcal{X}$ to $\mathcal{T}(\mathcal{C},\mathcal{F})$ (resp. $\mathcal{T}(\mathcal{C})$).

We now give the semantics of confluent programs, that is programs for which the associated rewrite system is confluent. The domain of interpretation is the constructor algebra $\mathcal{T}(\mathcal{C})$.

Definition 4. *Let* `main` *be a confluent program. The function computed by* `main` *is the partial function $[\![\mathtt{main}]\!] : \mathcal{T}(\mathcal{C})^n \to \mathcal{T}(\mathcal{C})$ where n is the arity of* `main` *which is defined as follows. For all $u_i \in \mathcal{T}(\mathcal{C})$, $[\![\mathtt{main}]\!](u_1, \cdots, u_n) = v$ iff $\mathtt{main}(u_1, \cdots, u_n) \xrightarrow{*} v$ with $v \in \mathcal{T}(\mathcal{C})$. Note that due to the form of the rules a constructor term is a normal form ; as the program is confluent, it is uniquely defined. Otherwise, that is if there is no such normal form, $[\![\mathtt{main}]\!](u_1, \cdots, u_n)$ is undefined.*

[1] We shall use type writer font for function symbol and bold face font for constructors.

3 LPO and Quasi-Interpretations

3.1 Lexicographic Path Ordering

Termination orderings are widely used to prove the termination of term rewriting systems. The *Lexicographic Path Ordering* (LPO) is one of them, it was introduced by Kamin and Lévy [15]. We briefly describe it, together with some basic properties we shall use later on.

Definition 5. *Let $\prec$ be a term ordering. We note $\prec^l$ its lexicographic extension. A precedence $\preceq_{\mathcal{F}}$ (strict precedence $\prec_{\mathcal{F}}$) is a quasi-ordering (ordering) on the set $\mathcal{F}$ of function symbols. It is canonically extended on $\mathcal{C} \cup \mathcal{F}$ by saying that constructors are smaller than functions. Given such a precedence, the lexicographic path ordering $\prec_{lpo}$ is defined recursively by the rules:*

$$\frac{s \preceq_{lpo} t_i}{s \prec_{lpo} f(\ldots,t_i,\ldots)}\, f \in \mathcal{F} \bigcup \mathcal{C} \qquad \frac{s_i \prec_{lpo} f(t_1,\cdots,t_n) \qquad g \prec_{\mathcal{F}} f}{g(s_1,\cdots,s_m) \prec_{lpo} f(t_1,\cdots,t_n)}\, g \in \mathcal{F} \bigcup \mathcal{C}$$

$$\frac{(s_1,\cdots,s_n) \prec^l_{lpo} (t_1,\cdots,t_n) \qquad f \approx_{\mathcal{F}} g \qquad s_j \prec_{lpo} f(t_1,\cdots,t_n)}{g(s_1,\cdots,s_n) \prec_{lpo} f(t_1,\cdots,t_n)}$$

Definition 6. *$\triangleleft$ is the usual subterm ordering. That is $s \triangleleft f(t_1,\cdots,t_n)$ if and only if $s = t_i$ or $s \triangleleft t_i$ for some $1 \leq i \leq n$.*

Lemma 7. *Let t and s be constructor terms. $s \prec_{lpo} t$ if and only if $s \triangleleft t$.*

Example 8. *One can verify that the Ackermann's function of example 3 terminates by LPO.*

3.2 Polynomial Quasi-Interpretation

Definition 9. *Let $f \in \mathcal{F} \bigcup \mathcal{C}$ be either a function symbol or a constructor of arity n. A quasi-interpretation of f is a mapping $(\!|f|\!) : \mathbf{N}^n \to \mathbf{N}$ which satisfies (i) $(\!|f|\!)$ is (not necessarily strictly) increasing with respect to each argument, (ii) $(\!|f|\!)(X_1,\cdots,X_n) \geq X_i$, for all $1 \leq i \leq n$, (iii) $(\!|f|\!) > 0$ for each 0-ary symbol $f \in \mathcal{F} \cup \mathcal{C}$.*

We extend a quasi-interpretation $(\!|-|\!)$ to terms canonically: $(\!|f(t_1,\cdots,t_n)|\!) = (\!|f|\!)((\!|t_1|\!),\cdots,(\!|t_n|\!))$

Definition 10. *$(\!|-|\!)$ is a quasi-interpretation of a program $\mathtt{main}$ if for each rule $l \to r \in \mathcal{E}(\mathtt{main})$ and for each closed substitution σ, $(\!|r\sigma|\!) \leq (\!|l\sigma|\!)$.*

Lemma 11. *If t and t' are two terms such that $t \to t'$, then $(\!|t|\!) \geq (\!|t'|\!)$.*

Definition 12. *A program $\mathtt{main}$ admits a polynomial quasi-interpretation $(\!|-|\!)$, if $(\!|-|\!)$ is bounded by a polynomial.*

A polynomial quasi-interpretation is said to be of kind 0 *if for each constructor* $\mathbf{c}$, $(\!|\mathbf{c}|\!)(X_1,\cdots,X_n) = a + \sum_{i=1}^{n} X_i$ *for some constant* $a > 0$.

Remark 13. Quasi-interpretations are not sufficient to prove program termination. Indeed, the rule $\mathtt{f}(x) \to \mathtt{f}(x)$ admits a quasi-interpretation but doesn't terminate.

Unlike quasi-interpretation, a program interpretation satisfies to extra conditions: (i) $(\!|f|\!)(X_1,\cdots,X_n) > X_i$, (ii) $(\!|r\sigma|\!) < (\!|l\sigma|\!)$. Programs admitting an interpretation terminate. This sort of termination proof, by polynomial interpretations, was introduced by Lankford [16]. Bonfante, Cichon, Marion and Touzet [3] proved that programs admitting interpretation of kind 0 are computable in polynomial time.

Definition 14. *A LPO-program is a program that terminates by LPO.*

A $LPO^{Poly(0)}$-program is a LPO-program that admits a quasi-interpretation of kind 0.

Theorem 15 (main result). *The set of functions computed by $LPO^{Poly(0)}$-programs is exactly the set of funtion computable in polynomial space.*

Proof. It is a consequence of Theorem 24 and Theorem 32.

Examples 16.

1. *The Ackermann's function of example 3 doesn't admit a polynomial quasi-interpretation because $[\![Ack]\!]$ is not polynomially bounded.*

2. *The Quantified Boolean Formula (QBF) is the problem of the validity of a boolean formula with quantifiers over propositional variables. It is well-known to be* PSPACE *complete. Wlog, we restrict formulae to* $\neg, \vee, \exists$. *It can be solved by the following rules:*

$$\begin{array}{rcc} \mathit{main}(\phi) \rightarrow \mathit{ver}(\phi, \mathit{nil}) & \mathit{not}(\mathit{tt}) \rightarrow \mathit{ff} & 0 = 0 \rightarrow \mathit{tt} \\ \mathit{in}(x, \mathit{nil}) \rightarrow \mathit{ff} & \mathit{not}(\mathit{ff}) \rightarrow \mathit{tt} & S(x) = 0 \rightarrow \mathit{ff} \\ \mathit{in}(x, \mathit{cons}(a, l)) \rightarrow \mathit{or}(x = a, \mathit{in}(x, l)) & \mathit{or}(\mathit{tt}, x) \rightarrow \mathit{tt} & 0 = S(y) \rightarrow \mathit{ff} \\ & \mathit{or}(\mathit{ff}, x) \rightarrow x & S(x) = S(y) \rightarrow x = y \end{array}$$

In the next function, t is the set of variables whose value is tt.

$$\begin{array}{ll} \mathit{ver}(\mathit{Var}(x), t) \rightarrow \mathit{in}(x, t) & \mathit{ver}(\mathit{Or}(\phi_1, \phi_2), t) \rightarrow \mathit{or}(\mathit{ver}(\phi_1, t), \mathit{ver}(\phi_2, t)) \\ \mathit{ver}(\mathit{Not}(\phi), t) \rightarrow \mathit{not}(\mathit{ver}(\phi, t)) & \mathit{ver}(\mathit{Exists}(n, \phi), t) \rightarrow \mathit{or}(\mathit{ver}(\phi, \mathit{cons}(n, t)), \mathit{ver}(\phi, t)) \end{array}$$

These rules are ordered by LPO by putting $\{\mathit{not}, \mathit{or}, \_=\_\} \prec_{\mathcal{F}} \mathit{in} \prec_{\mathcal{F}} \mathit{ver} \prec_{\mathcal{F}} \mathit{main}$.
They admit the following quasi-interpretations:
- $(\!|\mathsf{c}|\!)(X_1, \cdots, X_n) = 1 + \sum_{i=1}^{n} X_i$, *for each n-ary constructor,*
- $(\!|\mathit{ver}|\!)(\Phi, T) = \Phi + T$, $(\!|\mathit{main}|\!)(\Phi) = \Phi + 1$,
- $(\!|f|\!)(X_1, \cdots, X_n) = \max_{i=1}^{n} X_i$, *for the other function symbols.*

4 LPO$^{Poly(0)}$-programs are PSPACE computable

Definition 17. *A* state *is a tuple* $\langle f, t_1, \cdots, t_n \rangle$ *where* f *is a function symbol of arity* n *and* $t_1, \ldots, t_n$ *are constructor terms.*

State(main) *is the set of all states built from the symbols of a program* main. $\mathit{State}^A(\mathit{main}) = \{\langle f, t_1, \cdots, t_n \rangle \in \mathit{State}(\mathit{main}) \ / \ |t_i| < A\}$. *Intutively, a state represents a recursive call in the evaluation process.*

Definition 18. *Let* main *be a LPO*$^{Poly(0)}$*-program,* $\eta_1 = \langle f, t_1, \cdots, t_n \rangle$ *and* $\eta_2 = \langle g, s_1, \cdots, s_m \rangle$ *be two states of State*(main). *A* transition *is a triplet* $\eta_1 \overset{e}{\rightsquigarrow} \eta_2$ *such that:*

(i) $e \equiv f(p_1, \cdots, p_n) \rightarrow t$
(ii) there is a substitution σ *such that* $p_i\sigma = t_i$ *for all* $1 \leq i \leq n$
(iii) there is a subterm $g(u_1, \cdots, u_m) \trianglelefteq t$ *such that* $u_i\sigma \overset{!}{\rightarrow} s_i$ *for all* $1 \leq i \leq n$.

Transition(main) *is the set of all transitions between the elements of State*(main).
$\overset{*}{\rightsquigarrow}$ *is the reflexive transitive closure of* $\cup_{e \in \mathcal{E}} \overset{e}{\rightsquigarrow}$.

Definition 19. *Let* $\mathcal{E}(\mathit{main})$ *be a LPO*$^{Poly(0)}$*-program and* $\langle f, t_1, \cdots, t_n \rangle \in \mathit{State}(\mathit{main})$ *be a state. A* $\langle f, t_1, \cdots, t_n \rangle$*-call tree* τ *is defined as follows:*

- *The root of* τ *is* $\langle f, t_1, \cdots, t_n \rangle$.
- *For each node* η_1, *the children of* η_1 *is exactly the set of states*

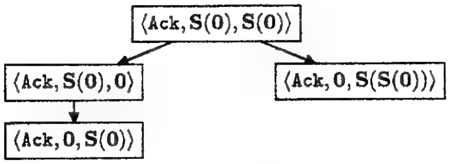
$$\{\eta_2 \in \mathit{State}(\mathit{main}) \ / \ \eta_1 \overset{e}{\rightsquigarrow} \eta_2 \in \mathit{Transition}(\mathit{main})\}$$

where e is a given equation of $\mathcal{E}$.

$CT(\langle f, t_1, \cdots, t_n \rangle)$ *is the set of all* $\langle f, t_1, \cdots, t_n \rangle$*-call trees.*

$$CT^A(\langle f, t_1, \cdots, t_n \rangle) = \{\tau \in CT(\langle f, t_1, \cdots, t_n \rangle) \ / \ \forall \eta \in \tau, \eta \in \mathit{State}^A(\mathit{main})\}$$

Example 20. *The (unique)* $\langle \mathit{Ack}, S(0), S(0) \rangle$*-call tree of* $CT(\langle \mathit{Ack}, S(0), S(0) \rangle)$ *is:*



Lemma 21. *Let* main *be a LPO-program,* α *be the number of function symbols in* main *and* d *be the maximal arity of a function symbol. The following facts hold for all* $\tau \in CT^A(\langle f, t_1, \cdots, t_n \rangle)$:

1. *If $\langle f, t_1, \cdots, t_n\rangle \overset{*}{\rightsquigarrow} \langle g, s_1, \cdots, s_m\rangle$ then (a) $g \prec_{\mathcal{F}} f$ or (b) $g \approx_{\mathcal{F}} f$ and $(s_1, \cdots, s_m) \prec^l_{lpo} (t_1, \cdots, t_n)$.*
2. *If $\langle f, t_1, \cdots, t_n\rangle \overset{*}{\rightsquigarrow} \langle g, s_1, \cdots, s_m\rangle$ in τ and $g \approx_{\mathcal{F}} f$ then the number of states between $\langle f, t_1, \cdots, t_n\rangle$ and $\langle g, s_1, \cdots, s_m\rangle$ is bounded by A^d.*
3. *The length of each branch of τ is bounded by $\alpha \times A^d$.*

Proof.

1. Because the rules of the program decrease for LPO.
2. Let $\langle \mathtt{h}, u_1, \cdots, u_p\rangle$ be a child of $\langle \mathtt{f}, t_1, \cdots, t_n\rangle$. As t_i and u_j are constructor terms, due to first point of the current lemma and lemma 7, we have $(s_1, \cdots, s_m) \triangleleft^l (t_1, \cdots, t_n)$. So, we may conclude $(|t_1|, \cdots, |t_n|) <^l (|u_1|, \cdots, |u_p|)$. Since the size of each component is bounded by A, the length of the decreasing chain is bounded by A^d.
3. In each branch, there are at most A^d states whose function symbols have the same precedence, then A^d states whose function symbol have the precedence immediatly below, and so on. As there are only α function symbols the length of the branch is bounded by $\alpha \times A^d$.

Lemma 22. *Let $main$ be a $LPO^{Poly(0)}$-program, f be a function symbol and $t_1, \cdots, t_n$ be constructor terms. $CT(f, t_1, \cdots, t_n) = CT^{\llparenthesis f(t_1, \cdots, t_n)\rrparenthesis}(f, t_1, \cdots, t_n)$.*

Proof. Let $\tau \in CT(\mathtt{f}, t_1, \cdots, t_n)$ and $\langle \mathtt{g}, s_1, \cdots, s_m\rangle$ be a state in τ. As s_i is a constructor term, $|s_i| \leq \llparenthesis s_i \rrparenthesis \leq \llparenthesis \mathtt{g}(s_1, \cdots, s_m)\rrparenthesis \leq \llparenthesis \mathtt{f}(t_1, \cdots, t_n)\rrparenthesis$ because of the definition of quasi-interpretations.

Lemma 23. *Given a term $t \in \mathcal{T}(\mathcal{C})$, the following holds: $\llparenthesis t \rrparenthesis \leq c.|t|$ for some constant c. As a corollary, we have $main(t_1, \ldots, t_n) \leq P(\max_{i=1}^n |t_i|)$ for some polynomial P.*

Theorem 24. *Let $main$ be a $LPO^{Poly(0)}$-program. For each constructor terms $t_1, \cdots, t_n$, the space used by a call by value interpreter to compute $main(t_1, \cdots, t_n)$ is bounded by a polynomial in $max_i\{|t_i|\}$. Such an interpreter is described in annex.*

Proof. Put $A = \llparenthesis \mathtt{main}(t_1, \cdots, t_n)\rrparenthesis$.

The interpreter only needs to store the call stack of each recursive call and the intermediate terms (e_i, b) of the computation. The size of e_i and b are both bounded by A. Note that the computation can be followed on a $\langle \mathtt{main}, t_1, \ldots, t_n\rangle$ call-tree. Each recursive call corresponds a transition on the call-tree. So, the maximal depth of the stack corresponds to the maximal length of the branch in a call tree of $CT\langle \mathtt{main}, t_1, \cdots, t_n\rangle = CT^A\langle \mathtt{main}, t_1, \cdots, t_n\rangle$. So it is bounded by $\alpha \times A^d$ (see Lemma 21(3)). The values stored in the stack are states; as a consequence, the size of each of them is bounded by $d \times A + O(1)$.

Therefore, the space used by the interpreter is bounded by $\alpha \times d \times A^{d+1} + A + O(1)$, and A is a polynomial in the size of $\max_i\{|t_i|\}$ by Lemma 23.

5 Parallel Register Machines over Words

We present here a restriction of Parallel Register Machines introduced in [18]. They are an adaptation of the alternating Turing machine. Chandra, Kozen and Stockmeyer have demonstrated that the set of functions that alternating Turing machines computes in polynomial time is exactly the state of polynomial space computable functions.

Definition 25. $\mathbb{W} = \mathcal{T}(\{0^1, 1^1, \epsilon^0\})$. $\mathbb{N} = \mathcal{T}(\{s^1, \diamond^0\})$.

Note that $\mathbb{W}$ (resp. $\mathbb{N}$) is isomorphic to $\{0, 1\}^*$ (resp. natural numbers), both sets are used indiferently in the rest of the section.

Definition 26. *A Parallel Register Machine (PRM) over the word algebra $\mathbb{W}$ consists in:*

1. *a finite set $S = \{s_0, s_1, \ldots, s_k\}$ of* states, *including a distinct state* BEGIN.
2. *a finite list $\Pi = \{\pi_1, \ldots, \pi_m\}$ of* registers; *we write* OUTPUT *for π_m; Registers will only store values in $\mathbb{W}$;*
3. *an ordering $\blacktriangleleft$ on $\mathbb{W}$: $\epsilon \blacktriangleleft y$, $0(x) \blacktriangleleft 1(y)$, $i(x) \blacktriangleleft i(y)$ if and only if $x \blacktriangleleft y$.*
4. *a function com mapping states to* commands *which are :[$\boldsymbol{Succ}(\pi' = \mathbf{i}(\pi), s')$], [$\boldsymbol{Pred}(\pi' = \mathbf{p}(\pi), s')$], [$\boldsymbol{Branch}(\pi, s', s'')$], [$\boldsymbol{Fork}_{min}(s', s'')$], [$\boldsymbol{Fork}_{max}(s', s'')$], [$\boldsymbol{End}$].*

A *configuration* of a PRM M is given by a pair (s, F) where $s \in S$ and F is a function $\Pi \rightarrow \mathbb{W}$. We note $[u_1, \ldots, u_m]$ for the function which maps $\pi_i \mapsto u_i$ and $\{\pi_i \leftarrow a\}[u_1, \ldots, u_m]$ denotes $[u_1, \ldots, u_{i-1}, a, u_{i+1}, \ldots, u_m]$.

Definition 27. *Given M as above we define a semantic partial-function $\mathit{eval} : \mathbb{N} \times S \times \mathbb{W}^m \mapsto \mathbb{W}$, that maps the result of the machine in a "time bound" given by the first argument.*

- *$\mathit{eval}(0, s, F)$ is undefined.*
- *If $com(s)$ is $\mathbf{Succ}(\pi' = \mathbf{i}(\pi), s')$ then $\mathit{eval}(t+1, s, F) = \mathit{eval}(t, s', \{\pi' \leftarrow \mathbf{i}(\pi)\}F)$. Note that on the right of the left arrow, π denotes the content of the register;*
- *If $com(s)$ is $\mathbf{Pred}(\pi' = \mathrm{p}(\pi), s')]$, then $\mathit{eval}(t+1, s, F) = \mathit{eval}(t, s', \{\pi' \leftarrow \mathrm{p}(\pi)\}F)$;*
- *If $com(s)$ is $\mathbf{Branch}(\pi, s', s'')$ then $\mathit{eval}(t+1, s, F) = \mathit{eval}(t, r, F)$, where $r = s'$ if $\pi = \mathit{0}(w)$ and $r = s''$ if $\pi = \mathit{1}(w)$;*
- *If $com(s)$ is $\mathbf{Fork}_{\min}(s', s'')$ then $\mathit{eval}(t+1, s, F) = \min_{\blacktriangleleft}(\mathit{eval}(t, s', F), \mathit{eval}(t, s'', F))$;*
- *If $com(s)$ is $\mathbf{Fork}_{max}(s', s'')$ then $\mathit{eval}(t+1, s, F) = \max_{\blacktriangleleft}(\mathit{eval}(t, s', F), \mathit{eval}(t, s'', F))$;*
- *If $com(s)$ is $\mathbf{End}$ then $\mathit{eval}(t+1, s, F) = F(\text{OUTPUT})$.*

Definition 28. *Given a function $T : \mathbb{N} \to \mathbb{N}$, we say that the PRM M computes $f : \mathbb{W}^k \to \mathbb{W}$ in time T (or equivalently that f is T-computable) if for all $(w_1, \cdots, w_k) \in \mathbb{W}^k$, we have*

$$\mathit{eval}(T(\max_{i=1}^{k} |w_i|), \text{BEGIN}, [w_1, \cdots, w_k, \epsilon, \ldots, \epsilon]) = f(w_1, \cdots, w_k)$$

Theorem 29 (Chandra & *al* [4]). *Let $f : \mathbb{W} \to \mathbb{W}$. f is computable in polynomial space iff f is PRM-computable in polynomial time.*

5.1 Simulation of PRMs by LPO$^{Poly(0)}$-programs

The simulation of PRM is done simply by following the rules of the operational semantics of PRM we gave above. In particular, the first argument of `eval` represents a clock.

Lemma 30 (Plug and play lemma). *Let $f : \mathbb{W} \to \mathbb{W}$ be a T-time PRM-computable function, then, the function f' is computable by an LPO$^{Poly(0)}$.*

$$\begin{aligned} f' : \mathbb{N} \times \mathbb{W} &\to \mathbb{W} \\ (n, w) &\mapsto f(w) \quad \text{if } n > T(|w|) \\ (n, w) &\mapsto \bot \qquad \text{otherwise} \end{aligned}$$

Proof. Let the set of constructors be $C = \{\mathbf{0}, \mathbf{1}, \mathbf{s}, \diamond, \epsilon\} \cup S$ where S is the set of states. We let the rules in appendix B. Simply, let's say that the functions symbols are: `min`, `max` corresponding to $\min_{\blacktriangleleft}, \max_{\blacktriangleleft}$, `eval` which simulate the rules of the operational semantics and $\mathtt{f}'$. We develop here two rules for `eval`.

- $\mathtt{Eval}(\mathbf{s}(t), s, \pi_1, \cdots, \pi_m) \to \mathtt{Eval}(t, s', \pi_1, \cdots, \pi_{j-1}, \mathbf{i}(\pi_k), \pi_{j+1}, \cdots, \pi_m)$ if $com(s) = \mathbf{Succ}(\pi_j = \mathbf{i}(\pi_k), s')$,
- $\mathtt{Eval}(\mathbf{s}(t), s, \pi_1, \cdots, \pi_m) \to \mathtt{min}(\mathtt{Eval}(t, s', \pi_1, \cdots, \pi_m), \mathtt{Eval}(t, s'', \pi_1, \cdots, \pi_m))$ if $com(s) = \mathbf{Fork}_{\min}(s', s'')$

Take the precedence $\{\mathtt{min}, \mathtt{max}\} \prec_{\mathcal{F}} \mathtt{Eval}$, these rules decrease according to LPO because the time bound decrease. They admit the following quasi-interpretation:

$$\begin{array}{lll} (\!|\epsilon|\!) = 1 & (\!|\mathbf{0}|\!)(X) = X + 1 & (\!|\mathtt{min}|\!)(W, W') = \max(W, W') \\ (\!|\diamond|\!) = 1 & (\!|\mathbf{1}|\!)(X) = X + 1 & (\!|\mathtt{max}|\!)(W, W') = \max(W, W') \\ & (\!|\mathbf{s}|\!)(X) = X + 1 & \end{array}$$

$$\forall s \in S, (\!|s|\!) = 1 \qquad (\!|\mathtt{Eval}|\!)(T, S, \Pi_1, \cdots, \Pi_m) = \max\{\Pi_1, \cdots, \Pi_m\} \times T + S$$

Now, the function $\mathtt{f}'$ is defined by $\mathtt{f}'(n, w) \to \mathtt{Eval}(n, \text{BEGIN}, w, \epsilon, \ldots, \epsilon)$. It is routine to check that $f' = [\![\mathtt{f}']\!]$. This rule decreases by LPO by taking $\mathtt{f}' \succ \mathtt{Eval}$. There is also a quasi-interpretation for the rule: $(\!|\mathtt{f}'|\!)(N, X) = N \times X + 1$.

Lemma 31. *Given $w \in \mathbb{W}$ and a polynomial P, the function $w \in \mathbb{W} \mapsto P(|w|)$ is computable by a LPO$^{Poly(0)}$-program. The proof is given in the appendix.*

Theorem 32. *A polynomial time PRM computable function f can be computed by an LPO$^{Poly(0)}$-program.*

Proof. Follows from Lemma 30 and Lemma 31.

References

1. S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2:97–110, 1992.
2. R. Benzinger. *Automated complexity analysis of NUPRL extracts.* PhD thesis, Cornell University, 1999.
3. G. Bonfante, A. Cichon, J-Y Marion, and H. Touzet. Complexity classes and rewrite systems with polynomial interpretation. In *Computer Science Logic, 12th International Workshop, CSL'98*, volume 1584 of *Lecture Notes in Computer Science*, pages 372–384, 1999.
4. A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *Journal of the ACM*, 28:114–133, 1981.
5. A. Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, *Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science*, pages 24–30. North-Holland, Amsterdam, 1962.
6. R. Constable and al. *Implementing Mathematics with the Nuprl Development System.* Prentice-Hall, 1986. `http://www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html`.
7. K. Crary and S. Weirich. Ressource bound certification. In *ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL*, pages 184 – 198, 2000.
8. N. Dershowitz and J-P Jouannaud. *Handbook of Theoretical Computer Science vol.B*, chapter Rewrite systems, pages 243–320. Elsevier Science Publishers B. V. (NorthHolland), 1990.
9. A. Goerdt. Characterizing complexity classes by higher type primitive recursive definitions. *Theoretical Computer Science*, 100(1):45–66, 1992.
10. D. Hofbauer. Termination proofs with multiset path orderings imply primitive recursive derivation lengths. *Theoretical Computer Science*, 105(1):129–140, 1992.
11. M. Hofmann. Linear types and non-size-increasing polynomial time computation. In *Proceedings of the Fourteenth IEEE Symposium on Logic in Computer Science (LICS'99)*, pages 464–473, 1999.
12. M. Hofmann. A type system for bounded space and functional in-place update. In *European Symposium on Programming, ESOP'00*, volume 1782 of *Lecture Notes in Computer Science*, pages 165–179, 2000.
13. N. Immerman. *Descriptive Complexity.* Springer, 1999.
14. N. Jones. The Expressive Power of Higher order Types or, Life without CONS. to appear, 2000.
15. S. Kamin and J-J Lévy. Attempts for generalising the recursive path orderings. Technical report, Univerity of Illinois, Urbana, 1980. Unpublished note.
16. D.S. Lankford. On proving term rewriting systems are noetherien. Technical Report MTP-3, Louisiana Technical University, 1979.
17. D. Leivant. Predicative recurrence and computational complexity I: Word recurrence and poly-time. In Peter Clote and Jeffery Remmel, editors, *Feasible Mathematics II*, pages 320–343. Birkhäuser, 1994.
18. D. Leivant and J-Y Marion. Ramified recurrence and computational complexity II: substitution and poly-space. In L. Pacholski and J. Tiuryn, editors, *Computer Science Logic, 8th Workshop, CSL '94*, volume 933 of *Lecture Notes in Computer Science*, pages 486–500, Kazimierz,Poland, 1995. Springer.
19. J-Y Marion. Complexité implicite des calculs, de la théorie à la pratique, 2000. Habilitation.
20. J-Y Marion and J-Y Moyen. Efficient first order functional program interpreter with time bound certifications. In *LPAR*, volume 1955 of *Lecture Notes in Computer Science*, pages 25–42. Springer, Nov 2000.
21. R. Nieuwenhuis. Simple LPO constraint solving methods. *Information Processing Letters*, 47:65–69, August 1993.
22. D.B. Thompson. Subrecursiveness : machine independent notions of computability in restricted time and storage. *Math. System Theory*, 6:3–15, 1972.
23. A. Weiermann. Termination proofs by lexicographic path orderings yield multiply recursive derivation lengths. *Theoretical Computer Science*, 139:335–362, 1995.

A Interpreter for LPO$^{Poly(0)}$-programs

A space-economical interpreter:

```
function eval($\mathtt{f}, t_1, \cdots, t_n$)
begin
    let $e = \mathtt{f}(p_1, \cdots, p_n) \to t$
    let $\sigma$ such that $p_i\sigma = t_i$
    let $e_0 = t\sigma$
    let $i = 0$
    foreach $\mathrm{g}(s_1, \cdots, s_m) \trianglelefteq e_i \wedge s_j \in \mathcal{T}(\mathcal{C}), j \in \{1,..,m\}$ do
        $b = \mathrm{eval}(\mathrm{g}, s_1, \cdots, s_m)$
        $e_{i+1} = e_i\{\mathrm{g}(s_1, \cdots, s_m) \leftarrow b\}$
        $i = i + 1$
    return $e_i$
end.
```

where $e_i\{\mathrm{g}(s_1, \cdots, s_m) \leftarrow b\}$ denotes the term e_i where each occurence of $\mathrm{g}(s_1, \cdots, s_m)$ has been replaced by b.

B Simulation of PRM by LPO$^{Poly(0)}$-programs

One follows the operational semantics of PRMs.

$$\begin{array}{rlrl}
\mathtt{min}(\epsilon, w) \to \epsilon & & \mathtt{max}(\epsilon, w) \to w \\
\mathtt{min}(w, \epsilon) \to \epsilon & & \mathtt{max}(w, \epsilon) \to w \\
\mathtt{min}(\mathbf{0}(w), \mathbf{1}(w')) \to \mathbf{0}(w) & & \mathtt{max}(\mathbf{0}(w), \mathbf{1}(w')) \to \mathbf{1}(w') \\
\mathtt{min}(\mathbf{1}(w), \mathbf{0}(w')) \to \mathbf{0}(w') & & \mathtt{max}(\mathbf{1}(w), \mathbf{0}(w')) \to \mathbf{1}(w) \\
\mathtt{min}(\mathbf{i}(w), \mathbf{i}(w')) \to \mathbf{i}(\mathtt{min}(w, w')) & & \mathtt{max}(\mathbf{i}(w), \mathbf{i}(w')) \to \mathbf{i}(\mathtt{max}(w, w'))
\end{array}$$

with $\mathbf{i} \in \{\mathbf{0}, \mathbf{1}\}$. We have $[\![\mathtt{min}]\!] = \min_{\blacktriangleleft}$ and $[\![\mathtt{max}]\!] = \max_{\blacktriangleleft}$.

(a) $\mathtt{Eval}(\mathtt{s}(t), s, \pi_1, \cdots, \pi_m) \to \mathtt{Eval}(t, s', \pi_1, \cdots, \pi_{j-1}, \mathbf{i}(\pi_k), \pi_{j+1}, \cdots, \pi_m)$
if $com(s) = \mathbf{Succ}(\pi_j = \mathbf{i}(\pi_k), s')$,

(b) $\mathtt{Eval}(\mathtt{s}(t), s, \pi_1, \ldots, \mathbf{i}(\pi'_j), \ldots, \pi_m) \to \mathtt{Eval}(t, s', \pi_1, \ldots, \pi'_j, \ldots, \mathbf{i}(\pi'_j), \ldots, \pi_m))$
if $com(s) = \mathbf{Pred}(\pi_k = \mathbf{p}(\pi_j), s')$,

(c) $\mathtt{Eval}(\mathtt{s}(t), s, \pi_1, \cdots, \pi_{j-1}, \mathbf{i}(\pi_j), \pi_{j+1}, \cdots, \pi_m) \to \mathtt{Eval}(t, r, \pi_1, \cdots, \pi_m)$
if $com(s) = \mathbf{Branch}(\pi_j, s', s'')$ where $r = s'$ if $\mathbf{i} = \mathbf{0}$ and $r = s''$ if $\mathbf{i} = \mathbf{1}$,

(d) $\mathtt{Eval}(\mathtt{s}(t), s, \pi_1, \cdots, \pi_m) \to \mathtt{min}(\mathtt{Eval}(t, s', \pi_1, \cdots, \pi_m), \mathtt{Eval}(t, s'', \pi_1, \cdots, \pi_m))$
if $com(s) = \mathbf{Fork}_{\mathrm{min}}(s', s'')$

(e) $\mathtt{Eval}(\mathtt{s}(t), s, \pi_1, \cdots, \pi_m) \to \mathtt{max}(\mathtt{Eval}(t, s', \pi_1, \cdots, \pi_m), \mathtt{Eval}(t, s'', \pi_1, \cdots, \pi_m))$
if $com(s) = \mathbf{Fork}_{\mathrm{max}}(s', s'')$

(f) $\mathtt{Eval}(\mathtt{s}(t), s, \pi_1, \ldots, \pi_m) \to \pi_m$
if $com(s) = \mathbf{End}$

B.1 Computation of Polynomials by LPO$^{Poly(0)}$-programs

Each polynomials can be computed with a combination of add and mult.

$$\begin{array}{rr}
\mathtt{add}(\diamond, y) \to y & \mathtt{mult}(\diamond, y) \to \diamond \\
\mathtt{add}(\mathtt{s}(x), y) \to \mathtt{s}(\mathtt{add}(x, y)) & \mathtt{mult}(\mathtt{s}(x), y) \to \mathtt{add}(y, \mathtt{mult}(x, y))
\end{array}$$

These functions are clearly ordered by LPO by putting $\mathtt{add} \prec_{\mathcal{F}} \mathtt{mult}$. And they admit the quasi-interpretations:

$$(\!|\mathtt{add}|\!)(X, Y) = X + Y$$
$$(\!|\mathtt{mult}|\!)(X, Y) = X \times Y$$

Remark 33. The interpretation of a polynomial correponds to its semantics.

Generalised Computability and Applications to Hybrid Systems*

Margarita V. Korovina[1] and Oleg V. Kudinov[2]

[1] A. P. Ershov Institute of Informatics Systems
Siberian Division of the Russian Academy of Sciences
6, Acad. Lavrentiev ave., 630090, Novosibirsk, Russia
e-mail: rita@inet.ssc.nsu.ru
[2] Institute of Mathematics
Koptug pr., 4, Novosibirsk, Russia
e-mail: kud@math.nsc.ru

Abstract. We investigate the concept of generalised computability of operators and functionals defined on the set of continuous functions, firstly introduced in [9]. By working in the reals, with equality and without equality, we study properties of generalised computable operators and functionals. Also we propose an interesting application to formalisation of hybrid systems. We obtain some class of hybrid systems, which trajectories are computable in the sense of computable analysis.

1 Introduction

Computability theories on particular and general classes of structures address central concerns in mathematics and computer science. The concept of generalised computability is closely related to definability theory investigated in [3]. This theory has many uses in computer science and mathematics because it can be applied to analyse computation on abstract structure, in particularly, on the real numbers or on the class of continuous functions. The main aim of our paper is to study properties of operators and functionals considering their generalised computability relative either to the ordered reals with equality, or to the strictly ordered real field. Note that generalised computability related to the strictly ordered real field is equivalent to computability in computable analysis in that they define the same class of computable real-valued functions and functionals. We prove that any continuous operator is generalised computable in the language with equality if and only it is generalised computable in the language without equality. As a direct corollary we obtain that each continuous generalised computable with equality operator is computable in the sense of computable analysis. This paper is structured as follows. In Section 2, we give basic definitions and tools. We study properties of operators and functionals considering their generalised computability in the language with equality and in the language without equality. In Section 3 we present some application of the proposed model of computation to specification of hybrid systems. In the recent time, attention to the problems of exact mathematical formalisation of complex systems such as hybrid systems is constantly raised. By a hybrid system we mean a network of digital and analog devices interacting at discrete times. An important characteristic of hybrid systems is that they incorporate both continuous components, usually called plants, as well as digital components, i.e. digital computers, sensors and actuators controlled by programs. These programs are designed to select, control, and supervise the behaviour of the continuous components. Modelling, design, and investigation of behaviours of hybrid systems have recently become active areas of research in computer science (for example see [5, 10, 12, 13]). The main subject of our investigation is behaviour of the continuous components. In [12], the set of all possible trajectories of the plant was called as a performance specification. Based on the proposed model of computation we introduce logical formalisation of hybrid systems in which the trajectories of the continuous components (the performance specification) are presented by computable functionals.

2 Generalised Computability

Throughout the article we consider two models of the real numbers,

$$<\mathbb{R},\sigma_1>\rightleftharpoons<\mathbb{R},0,1,+,\cdot,<,-x,\frac{x}{2}>$$

* This research was supported in part by the RFBR (grants N 99-01-00485, N 00-01-00810) and by the Siberian Division of RAS (a grant for young researchers, 2000)

is the model of the reals without equality, and

$$<\mathbb{R}, \sigma_2> \rightleftharpoons <\mathbb{R}, 0, 1, +, \cdot, \leq>$$

is the model of the reals with equality. Below if statements concern languages σ_1 and σ_2 we will write σ for a language.

Denote $\mathbf{D}_2 = \{z \cdot 2^{-n} | z \in \mathbb{Z},\ n \in \mathbb{N}\}$. Let us use $\bar{r}$ to denote $r_1, \dots, r_m$.

To recall the notion of generalised computability, let us construct the set of hereditarily finite sets $\mathrm{HF}(M)$ over a model $\mathbf{M}$. This structure is rather well studied in the theory of admissible sets [1] and permits us to define the natural numbers and to code and store information via formulas. Let $\mathbf{M}$ be a model of a language σ whose carrier set is M. We construct the set of hereditarily finite sets, $\mathrm{HF}(M) = \bigcup_{n\in\omega} \mathrm{S}_n(M)$, where $\mathrm{S}_0(M) \rightleftharpoons M$, $\mathrm{S}_{n+1}(M) \rightleftharpoons \mathcal{P}_\omega(\mathrm{S}_n(M)) \cup \mathrm{S}_n(M)$, where $n \in \omega$ and for every set B, $\mathcal{P}_\omega(B)$ is the set of all finite subsets of B.

We define $\mathbf{HF(M)} \rightleftharpoons \langle \mathrm{HF}(M), M, \sigma, \emptyset_{\mathbf{HF(M)}}, \in_{\mathbf{HF(M)}} \rangle$, where the unary predicate $\emptyset$ singles out the empty set and the binary predicate symbol $\in_{\mathbf{HF(M)}}$ has the set-theoretic interpretation.

Below we will consider $M \rightleftharpoons \mathbb{R}$, $\sigma_1^* = \sigma_1 \cup \{\in,\ \emptyset\}$ named the language without equality and $\sigma_2^* = \sigma_2 \cup \{\in,\ \emptyset\}$ named the language with equality.

To introduce the notions of terms and atomic formulas we use variables of two sorts. Variables of the first sort range over $\mathbb{R}$ and variables of the second sort range over $\mathrm{HF}(\mathbb{R})$.

The terms in the language σ_1^* are defined inductively by:

1. the constant symbols 0 and 1 are terms;
2. the variables of the first sort are terms;
3. if t_1, t_2 are terms then $t_1 + t_2$, $t_1 \cdot t_2$, $-t_1$, $\frac{t_2}{2}$ are terms.

The notions of a term in the language σ_2^* can be given in a similar way.

The following formulas in the language σ_1^* are atomic: $t_1 < t_2$, $t \in s$ and $s_1 \in s_2$ where t_1, t_2, t are terms and s_1, s_2 are variables of the second sort. The following formulas in the language σ_2^* are atomic: $t_1 < t_2$, $t \in s$, $s_1 \in s_2$ and $t_1 = t_2$ where t_1, t_2, t are terms and s_1, s_2 are variables of the second sort.

The set of *Δ_0-formulas* in the language σ^* is the closure of the set of atomic formulas in the language σ^* under $\wedge, \vee, \neg, (\exists x \in s)$ and $(\forall x \in s)$, where $(\exists x \in s)\ \varphi$ denotes $\exists x (x \in s \wedge\ \varphi)$ and $(\forall x \in s)\ \varphi$ denotes $\forall x (x \in s \to \varphi)$ and s is any variable of second type.

The set of *Σ-formulas* in the language σ^* is the closure of the set of Δ_0 formulas in the language σ^* under $\wedge, \vee, (\exists x \in s), (\forall x \in s)$, and $\exists$. The natural numbers $0,\ 1, \dots$ are identified with $\emptyset,\ \{\emptyset, \{\emptyset\}\}, \dots$ so that, in particular, $n + 1 = n \cup \{n\}$ and the set ω is a subset of $\mathrm{HF}(\mathbb{R})$.

Definition 1. *A relation $B \subseteq \mathbb{R}^n$ is Σ-definable in σ^*, if there exists a Σ-formula $\Phi(\bar{x})$ in the language σ^* such that $\bar{x} \in B \leftrightarrow \mathbf{HF}(\mathbb{R}) \models \Phi(\bar{x})$. A function is Σ-definable if its graph is Σ-definable.*

Note that the set $\mathbb{R}$ is Δ_0–definable in the language σ^*. This fact makes $\mathbf{HF}(\mathbb{R})$ a suitable domain for studying relations in $\mathbb{R}^n$ and functions from $\mathbb{R}^n$ to $\mathbb{R}$ where $n \in \omega$. For properties of Σ-definable relations in $\mathbb{R}^n$ we refer to [3, 6].

Without loss of generality we consider the set of continuous functions defined on the compact interval $[0, 1]$. To introduce generalised computability of operators and functionals we extend σ_1^* and σ_2^* by two 3-ary predicates U_1 and U_2.

Definition 2. *Let $\varphi_1(U_1, U_2, x_1, x_2, c)$, $\varphi_2(U_1, U_2, x_1, x_2, c)$ be formulas of extended language σ_1^* (σ_2^*). We suppose that U_1, U_2 occur positively in φ_1, φ_2 and the predicates U_1, U_2 define open sets in $\mathbb{R}^3$. The formulas φ_1, φ_2 are said to satisfy joint continuity property if the following formulas are valid in* $\mathbf{HF}(\mathbb{R})$.

1. $\forall x_1 \forall x_2 \forall x_3 \forall x_4 \forall z\, ((x_1 \leq x_3) \wedge (x_4 \leq x_2) \wedge \varphi_i(U_1, U_2, x_1, x_2, z)) \to$ $\varphi_i(U_1, U_2, x_3, x_4, z)$, *for* $i = 1, 2$
2. $\forall x_1 \forall x_2 \forall c \forall z\, ((z < c) \wedge \varphi_1(U_1, U_2, x_1, x_2, c)) \to \varphi_1(U_1, U_2, x_1, x_4, z)$,
3. $\forall x_1 \forall x_2 \forall c \forall z\, ((z > c) \wedge \varphi_2(U_1, U_2, x_1, x_2, c)) \to \varphi_2(U_1, U_2, x_1, x_4, z)$,
4. $\forall x_1 \forall x_2 \forall x_3 \forall z\, (\varphi_i(U_1, U_2, x_1, x_2, z) \wedge \varphi_i(U_1, U_2, x_2, x_3, z)) \to$ $\varphi_i(U_1, U_2, x_1, x_3, z)$, *for* $i = 1, 2$,
5. $(\forall y_1 \forall y_2 \exists z \forall z_1 \forall z_2 (U_1(y_1, y_2, z_1) \wedge U_2(y_1, y_2, z_1) \to (z_1 < z < z_2))) \to (\forall x_1 \forall x_2 \exists c \forall c_1 \forall c_2 (\varphi_1(U_1, U_2, x_1, x_2, c_1) \wedge$ $\varphi_2(U_1, U_2, x_1, x_2, c_2) \to$ $(c_1 < c < c_2)))$.

Definition 3. *A total operator $F : C[0,1] \to C[0,1]$ is said to be shared by two Σ-formulas φ_1 and φ_2 in the language σ^* if the following assertions hold. If $F(u) = h$ then $h|_{[x_1,x_2]} > z \leftrightarrow \mathbf{HF}(\mathbb{R}) \models \varphi_1(U_1, U_2, x_1, x_2, z)$ and $h|_{[x_1,x_2]} < z \leftrightarrow \mathbf{HF}(\mathbb{R}) \models \varphi_2(U_1, U_2, x_1, x_2, z)$, where $U_1(x_1, x_2, c) \rightleftharpoons u|_{[x_1,x_2]} > c, U_2(x_1, x_2, c) \rightleftharpoons u|_{[x_1,x_2]} < c$ and U_1, U_2 occur positively in φ_1, φ_2.*

Definition 4. *A total operator $F : C[0,1] \to C[0,1]$ is said to be generalised computable in the language σ^*, if F is shared by two Σ-formulas in the language σ^* which satisfy the joint continuity property.*

Definition 5. *A total functional $F : C[0,1] \times [0,1] \to \mathbb{R}$ is said to be generalised computable in the language σ^*, if there exists an operator $F^* : C[0,1] \to C[0,1]$ generalised computable in the language σ^* such that $F(f,x) = F^*(f)(x)$.*

Definition 6. *A total functional $F : C[0,1] \times \mathbb{R} \to \mathbb{R}$ is said to be generalised computable in the language σ^*, if there exists an effective sequence $\{F_n^*\}_{n\in\omega}$ of operators generalised computable in the language σ^* of the types $F_n^* : C[0,1] \to C[-n,n]$ such that $F(f,x) = y \leftrightarrow \forall n\,(-n \le x \le n \to F_n^*(f)(x) = y)$.*

Proposition 1. *A total functional $F : C[0,1] \times \mathbb{R} \to \mathbb{R}$ is generalised computable in the language without equality if and only if it is computable in the sense of computable analysis.*

Proof. See [9,17].

Now we propose the main theorem which connects generalised computabilities in the various languages.

Theorem 1. *A continuous total operator $F : C[0,1] \to C[0,1]$ is generalised computable in the language with equality if and only if it is generalised computable in the language without equality.*

Proposition 2. *Let $F : C[0,1] \times \mathbb{R} \to \mathbb{R}$ be a continuous total functional. The functional F is generalised computable in the language with equality if and only if it is generalised computable in the language without equality.*

Corollary 1. *Let $F : C[0,1] \times \mathbb{R} \to \mathbb{R}$ be a continuous functional. The functional F is generalised computable if and only if F is computable in the sense of computable analysis.*

Now we point attention to a useful recursion scheme, which permits us to describe the behavior of complex systems such as hybrid systems.

Let $\mathcal{F} : C[0,1] \times C[0,1] \times \mathbb{R} \to \mathbb{R}$ and $G : C[0,1] \times [0,1] \to \mathbb{R}$ be generalised computable in the language with equality functionals. Then $F : C[0,1] \times [0,+\infty) \to \mathbb{R}$ is defined by the following scheme:

$$\begin{cases} F(f,t)|_{t\in[0,1]} = G(f,t), \\ F(f,t)|_{t\in(n,n+1]} = \mathcal{F}(f,t,\lambda y F(f,y+n-1)) \end{cases}$$

Proposition 3. *The functional F is generalised computable in the language with equality, with F defined above.*

3 An Application to Formalisation of Hybrid Systems

We use the models of hybrid systems proposed by Nerode, Kohn in [12]. A hybrid system is a system which consists of a continuous plant that is disturbed by external world and controlled by a program implemented on a sequential automaton. The main subject of our investigation is behaviour of the continuous components. We propose a logical formalisation of hybrid systems in which the trajectories of the continuous components (the performance specification) are presented by computable functionals.

A formalisation of the hybrid system $\mathbf{FHS} = \langle TS, \mathbf{F}, Conv1, A, Conv2, I\rangle$ consists of:

- $TS = \{t_i\}_{i\in\omega}$. It is an effective sequence of rational numbers i.e. the set of Gordel numbers of elements of TS is computable enumerated. The rational numbers t_i are the times of communication of the external world and the hybrid system, and communication of the plant and the control automaton. The time sequence $\{t_i\}_{i\in\omega}$ satisfies the realizability requirements:
  1. For every i, $t_i \ge 0$; $t_0 < t_1 < \ldots < t_i \ldots$;
  2. The differences $t_{i+1} - t_i$ have positive lower bounds.
- $\mathbf{F} : C[0,1] \times \mathbb{R}^n \to \mathbb{R}$. It is a generalised computable functional in the language σ_2^*. The plant has been given by this functional.

- $Conv1 : C[0,1] \times \mathbb{R} \to A^*$. It is an generalised computable functional in the following sense: $Conf1(f,x) = w \Leftrightarrow \varphi(U_1, U_2, x, z)$, where $U_1(x_1,x_2,c) \rightleftharpoons f|_{[x_1,x_2]} > c, U_2(x_1,x_2,c) \rightleftharpoons f|_{[x_1,x_2]} < c$ and the predicate U_1 and U_2 occur positively in Σ-formula φ. At the time of communication this functional converts measurements, presented by the meaning of $\mathbf{F}$, and the representation of external world f into finite words which are input words of the internal control automata.
- $A : A^* \to A^*$. It is a Σ-definable function with parameters. The internal control automata, in practice, is a finite state automata with finite input and finite output alphabets. So, it is naturally modelled by Σ-definable function (see [3, 6]) which has a symbolic representation of measurements as input and produces a symbolic representation of the next control law as output.
- $Conv2 : A^* \to \mathbb{R}^{n-1}$. It is a Σ-definable function. This function converts finite words representing control laws into control laws imposed on the plant.
- $I \subset A^* \cup \mathbb{R}^n$. It is a finite set of initial conditions.

Definition 7. *The behaviour of a hybrid system is defined by a functional* $H : C[0,1] \times \mathbb{R} \to \mathbb{R}$ *if for any external disturbation* $f \in C[0,1]$ *the values of* $H(f,\cdot)$ *define the trajectory of the hybrid system.*

Theorem 2. *Suppose a hybrid system is specified as above. If the behaviour of the hybrid system is defined by a continuous functional H then H is computable in the sense of computable analysis.*

Proof. The claim follows from Theorem 1 and Proposition 3. □

In conclusion we would like to note that subjects of future papers will be formulation and investigation of optimal hybrid control in terms of generalised computability.

References

1. J. Barwise, Admissible sets and structures, Berlin, Springer–Verlag, 1975.
2. A. Edalat, P. Sünderhauf, A domain-theoretic approach to computability on the real line, Theoretical Computer Science, 210, 1998, pages 73–98.
3. Yu. L. Ershov, Definability and computability, Plenum, New York, 1996.
4. A. Grzegorczyk, On the definitions of computable real continuous functions, Fund. Math., N 44, 1957, pages 61–71.
5. T.A. Henzinger, Z. Manna, A. Pnueli, Towards refining Temporal Specifications into Hybrid Systems, LNCS N 736, 1993, pages 36–60.
6. M. Korovina, O. Kudinov, A New Approach to Computability over the Reals, SibAM, v. 8, N 3, 1998, pages 59–73.
7. M. Korovina, O. Kudinov, Characteristic Properties of Majorant-Computability over the Reals, Proc. of CSL'98, LNCS, 1584, 1999, pages 188–204.
8. M. Korovina, O. Kudinov, A Logical approach to Specifications of Hybrid Systems, Proc. of PSI'99, LNCS 1755, 2000, pages 10–16.
9. M. Korovina, O. Kudinov, Formalisation of Computability of Operators and Real-Valued Functionals via Domain Theory, Proceedings of CCA-2000, to appear in LNCS, 2001.
10. Z. Manna, A. Pnueli, Verifying Hybrid Systems, LNCS N 736, 1993, pages 4–36.
11. R. Montague, Recursion theory as a branch of model theory, Proc. of the third international congr. on Logic, Methodology and the Philos. of Sc., 1967, Amsterdam, 1968, pages 63–86.
12. A. Nerode, W. Kohn, Models for Hybrid Systems, Automata, Topologies, Controllability, Observability, LNCS N 736, 1993, pages 317–357.
13. W. Kohn, A. Nerode, J. B. Remmel Agent Based Velocity Control of Highway Systems, LNCS N 1273, 1997, pages 174–215.
14. M. B. Pour-El, J. I. Richards, Computability in Analysis and Physics, Springer-Verlag, 1988.
15. D. Scott, Outline of a mathematical theory of computation, In 4th Annual Princeton Conference on Information Sciences and Systems, 1970, pages 169–176.
16. Viggo Stoltenberg-Hansen, John V. Tucker, Concrete models of computation for topological algebras, TCS 219, 1999, pages 347-378.
17. K. Weihrauch, Computable analysis. An introduction, Springer, 2000.

Language Implementation

Exploring Template Template Parameters

Roland Weiss and Volker Simonis

Wilhelm-Schickard-Institut für Informatik, Universität Tübingen
Sand 13, 72076 Tübingen, Germany
e-mail: {weissr,simonis}@informatik.uni-tuebingen.de

Abstract. The generic programming paradigm has exerted great influence on the recent development of C++, e.g., large parts of its standard library [2] are based on generic containers and algorithms. While templates, the language feature of C++ that supports generic programming, have become widely used and well understood in the last years, one aspect of templates has been mostly ignored: template template parameters ([2], 14.1). In the first part, this article will present an in depth introduction of the new technique. The second part introduces a class for arbitrary precision arithmetic, whose design is based on template template parameters. Finally, we end with a discussion of the benefits and drawbacks of this new programming technique and how it applies to generic languages other than C++.

1 Introduction

The C++ standard library incorporated the standard template library (STL) [15] and its ideas, which are the cornerstones of generic programming [14]. Templates are the language feature that supports generic programming in C++. They come in two flavors, class templates and function templates. Class templates are used to express classes parameterized with types, e.g., the standard library containers, which hold elements of the argument type. Generic algorithms can be expressed with function templates. They allow one to formulate an algorithm independently of concrete types, such that the algorithm is applicable to a range of types complying to specific requirements. For example, the standard `sort` algorithm without function object ([2], 25.3) is able to rearrange a sequence of arbitrary type according to the order implied by the comparison operator <. Of course, the availability of this operator is a requirement on the elements' type.

It is possible to use instantiated class templates as arguments for class and function templates, therefore one is able to write nested constructs like `std::vec- tor <std::list<long> >`. So where does the need for template template parameters arise? Templates give one the power to abstract from an implementation detail, the types of the application's local data. Template template parameters provide one with the means to introduce an additional level of abstraction. Instead of using an instantiated class template as argument, the class template itself can be used as template argument. To clarify the meaning of this statement, we will look in the following sections at class and function templates that take template template parameters. Then we will present a generic arbitrary precision arithmetic implemented with template template parameters. Finally, the presented technique is discussed and effects on other generic languages are considered.

2 Class Templates

The standard library offers three sequence containers, `vector`, `list` and `deque`. They all have characteristics that recommend them for a given application context. But if one wants to write a new class called `store` that uses a standard container internally to store values, it is hard to choose the *perfect* container for all possible scenarios. This is exactly the situation where template template parameters fit in. The class designer can provide a default container, but the user can override this decision easily. Note that the user can not only use standard containers but also any proprietary container that conforms to the standard sequence container interface. Let us look at a code example that implements the class `store_comp` using object composition.

```
template < typename val_t,
  template <typename T, typename A> class cont_t = std::deque,
  typename alloc_t = std::allocator<val_t> >
class store_comp
{
  cont_t<val_t, alloc_t> m_cont; //instantiate template template parameter
public:
  typedef typename cont_t<val_t, alloc_t<val_t> >::iterator iterator;
  iterator begin() { return m_cont.begin(); }
  // more delegated methods...
};
```

The first template parameter `val_t` is the type of the objects to be kept inside the store. `cont_t`, the second one, is the template template parameter, which we are interested in. The declaration states that `cont_t` expects two template parameters `T` and `A`, therefore any standard conforming sequence container is applicable. We also provide a default value for the template template parameter, the standard container `deque`. When working with template template parameters, one has to get used to the fact that one provides a real class template as template argument, not an instantiation. The container's allocator `alloc_t` defaults to the standard allocator.

There is nothing unusual about the usage of `cont_t`, the private member `m_cont` is an instantiation of the default or user provided sequence container. As already mentioned, this implementation of `store_comp` applies composition to express the relationship between the new class and the internally used container. Another way to reach the same goal is to use inheritance, as shown in the following code segment:

```
template <typename val_t, ...>
class store_inh : public cont_t<val_t, alloc_t<val_t> > {};
```

The template header is the same as in the previous example. Due to the public inheritance, the user can work with the container's typical interface to change the store's content. For the class `store_comp`, appropriate member functions must be written, which delegate the actual work to the private member `m_cont`. The two differing designs of class `store` are summarized in Figure 1. The notation follows the diagrams in [9]. The only extension is that template template parameters inside the class' parameter list are typeset in boldface.

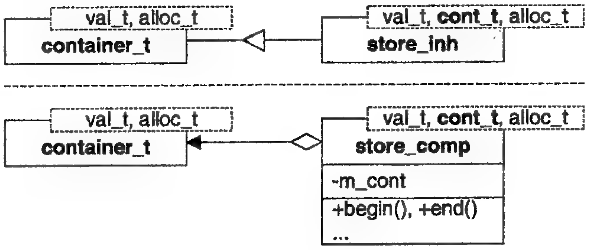


Fig. 1. Comparison of the competing designs of the `store` classes.

To conclude the overview, these code lines show how to create instances of the `store` classes:

```
store_comp<std::string, std::list> sc;
store_inh<int> si;
```

`sc` uses a `std::list` as internal container, whereas `si` uses the default container `std::deque`. This is a very convenient way for the user to select the appropriate container that matches the needs in his application area. The template template parameter can be seen as a container *policy* [1].

Now that we have seen how to apply template template parameters to a parameterized class in general, let us examine some of the subtleties.

First, the template template parameter – `cont_t` in our case – must be introduced with the keyword `class`, `typename` is not allowed ([2], 14.1). This makes sense, since a template template argument must correspond to a class template, not just a simple type name.

Also, the identifiers T and A introduced in the parameter list of the template temolate parameter are only valid inside its own declaration. Effectively, this means that they are not available inside the scope of the class store. One can instantiate the template template parameter inside the class body with different arguments multiple times, which would render the identifier(s) ambiguous. Hence, this scoping rule is reasonable.

But the most important point is the number of parameters of the template template parameter itself. Some of you may have wondered why two type parameters are given for a standard container, because they are almost exclusively instantiated with just the element type as argument, e.g., std::deque<float>. In these cases, the allocator parameter defaults to the standard allocator. Why do we have to declare it for cont_t? The answer is obvious: the template parameter signatures of the following two class templates C1 and C2 are distinct, though some of their instantiations can look the same:

```
template <typename T> class C1 {};
template <typename T1, typename T2 = int> class C2 {};
C1<double> c1; // c1 has signature C1<double>
C2<double> c2; // c2 has signature C2<double, int>
```

In order to be able to use standard containers, we have to declare cont_t conforming to the standard library. There ([2], 23.2), all sequence containers have two template parameters.[1] This can have some unexpected consequences. Think of a library implementor who decides to add another default parameter to a sequence container. Normal usage of this container is not affected by this implementation detail, but the class store can not be instantiated with this container because of the differing number of template parameters. We have encountered this particular problem with the deque implementation of the SGI STL [23].[2] Please note that some of the compilers that currently support template template parameters fail to check the number of arguments given to a template template parameter instantiation.

The template parameters of a template template parameter can have default arguments themselves. For example, if one is not interested in parameterizing a container by its allocator, one can provide the standard allocator as default argument and instantiate the container with just the contained type.

Finally, we will compare the approach with template template parameters to the traditional one using class arguments with template parameters. Such a class would look more or less like this:

```
template <typename cont_t>
class store_t
{
  cont_t m_cont; // use instantiated container for internal representation
public:
  typedef typename cont_t::iterator iterator;             // iterator type
  typedef typename cont_t::value_type value_type;         // value type
  typedef typename cont_t::allocator_type allocator_type; // alloc type
  // rest analogous to store_comp ...
};
typedef std::list<int> my_cont; // container for internal representation
store_t<my_cont> st;            // instantiate store
```

We will examine the advantages and drawbacks of each approach. The traditional one provides an instantiated class template as template argument. Therefore, store_t can extract all necessary types like the allocator, iterator etc. This is not possible in classes with template template parameters, because they perform the instantiation of the internal container themselves.

But the traditional approach was made applicable at all by the fact that the user provides the type with which the sequence container is instantiated. If the type is an implementation detail not made explicit to the user, the traditional approach doesn't work. See [21] for an application example with these properties: The ability to create multiple, different instantiations inside the class template body using the template template argument is also beyond the traditional approach:

```
cont_t<int, alloc_t> cont_1;
cont_t<val_t, std::allocator<val_t> > cont_2;
```

3 Function Templates

In the preceding section we showed that by application of template template parameters we gain flexibility in building data structures on top of existing STL container class templates. Now we want to examine what kind

[1] The C++ Standardization Committee currently discusses if this a defect, inadequately restricting library writers.

[2] The additional third template parameter was removed recently.

of abstractions are possible for generic functions with template template parameters. Of course, one can still use template template parameters to specify a class template for internal usage. This is analogous to the class store_comp, where object composition is employed.

But let us try to apply a corresponding abstraction to generic functions as we did to generic containers. We were able to give class users a convenient way to customize a complex data structure according to their application contexts. Transferring this abstraction to generic functions, we want to provide functions whose behavior is modifiable by their template template arguments.

We will exemplify this by adding a new method view to the class store. Its purpose is to print the store's content in a customizable way. A bare bones implementation inside a class definition is presented here:

```
template <template <typename iter_t> class mutator>
void view(std::ostream& os)
{
  mutator<iterator>()(begin(),end()); // iterator: defined in the store
  std::copy(begin(), end(), std::ostream_iterator<val_t>(os, " "));
}
```

Here, mutator is the template template parameter, it has an iterator type as template parameter. The mutator changes the order of the elements that are delimited by the two iterator arguments and then prints the changed sequence. This behavior is expressed in the two code lines inside the method body. The first line instantiates the mutator with the store's iterator and invokes the mutator's application operator, where the elements are rearranged. In the second line, the mutated store is written to the given output stream os, using the algorithm copy from the standard library. The types iterator and val_t are defined in the store class.

The first noteworthy point is that we have to get around an inherent problem of C++: functions are not first order objects. Fortunately, the same workaround already applied to this problem in the STL works fine. The solution is to use function objects (see [15], chapter 8). In the view method above, a function object that takes two iterators as arguments is required.

The following example shows how to write a function object that encapsulates the random_shuffle standard algorithm and how to call view with this function object as the mutator:

```
// function object that encapsulates std::random_shuffle
template <typename iter_t>
struct RandomShuffle
{
  void operator()(iter_t i1, iter_t i2) { std::random_shuffle(i1, i2); }
};
// A store s must be created and filled with values...
s.view<RandomShuffle>(cout); //RandomShuffle is the mutator
```

There are two requirements on the template arguments such that the presented technique works properly. First, the application operator provided by the function object, e.g., RandomShuffle, must match the usage inside the instantiated class template, e.g., store_comp. The view method works fine with application operators that expect two iterators as input arguments, like the wrapped random_shuffle algorithm from the standard library.

The second requirement touches the generic concepts on which the STL is built. RandomShuffle wraps the random_shuffle algorithm, which is specified to work with random access iterators. But what happens if one instantiates the store class template with std::list as template template argument and calls view<RandomShuffle>? std::list supports only bidirectional iterators, therefore the C++ compiler must fail instantiating view<RandomShuffle>. If one is interested in a function object that is usable with all possible store instantiations, two possibilities exist. Either we write a general algorithm and demand only the weakest iterator category, possibly loosing efficiency. Or we apply a technique already used in the standard library. The function object can have different specializations, which dispatch to the most efficient algorithm based on the iterator category. See [4] for a good discussion of this approach. This point, involving iterator and container categories as well as algorithm requirements, emphasizes the position of Musser et. al. [16] that generic programming is requirement oriented programming.

Completing, we want to explain why template template parameters are necessary for the view function and simple template parameters won't suffice. The key point is that the mutator can only be instantiated with the correct iterator. But the iterator is only know to the store, therefore an instantiation outside the class template store is not possible, at least not in a consistent manner.

Overall, the presented technique gives a class or library designer a versatile tool to make functions customizable by the user.

4 Long Integer Arithmetic — An Application Example

Now we will show how the techniques introduced in the last two sections can be applied to a real world problem. Suppose you want to implement a library for arbitrary precision arithmetic. One of the main problems one encounters is the question of how to represent long numbers. There are many well known possibilities to choose from: arrays, single linked lists, double linked lists, garbage collected or dynamically allocated and freed storage and so on. It is hard to make the right decision at the beginning of the project, especially because our decision will influence the way we have to implement the algorithms working on long numbers. Furthermore, we might not even know in advance all the algorithms that we eventually want to implement in the future.

The better way to go is to leave this decision open and parameterize the long number class by the container, which holds the digits. We just specify a minimal interface where every long number is a sequence of digits, and the digits of every sequence have to be accessible through iterators. With this in mind, we can define our long number class as follows:

```
template<
  template<typename T, typename A = std::allocator<T> >
    class cont_t = std::vector,
  template<typename AllocT> class alloc_t = std::allocator
>
class Integer {
  // ..
};
```

The first template template parameter stands for an arbitrary container type, which fulfills the requirements of a STL container. As we do not want to leave the memory management completely in the container's responsibility, we use a second template template parameter, which has the same interface as the standard allocator. Both template template parameters have default parameters, namely the standard vector class `std::vector` for the container and the standard allocator std::allocator for the allocator.

Knowing only this interface, a user could create `Integer` instances, which use different containers and allocators to manage a long number's digits. He even does not have to know if we use composition or inheritance in our implementation (see Figure 1 for a summary of the two design paradigms).[3]

In order to give the user access to the long number's digits, we implement the methods `begin()`, `end()` and `push_back()`, which are merely wrappers to the very same methods of the parameterized container. The first two return iterators that give access to the actual digits while the last one can be used to append a digit at the end of the long number. Notice that the type of a digit is treated as an implementation detail. We only have to make it available by defining a public type called `digit_type` in our class. Also we hand over in this way the type definitions of the iterators of the underlying containers. Now, our augmented class looks as follows (with the template definition omitted):

```
class Integer {
public:
  typedef int digit_type;
  typedef typename cont_t::iterator iterator;
  iterator begin() { return cont->begin(); }
  iterator end() { return cont->end(); }
  void push_back(digit_type v) { cont->push_back(v); }
private:
  cont_t<digit_type, alloc_t> *cont;
};
```

With this in mind and provided addition is defined for the digit type, a user may implement a naive addition without carry for long numbers of equal length in the following way (again the template definition has been omitted):

```
Integer<cont_t, alloc_t>
add(Integer<cont_t, alloc_t> &a, Integer<cont_t, alloc_t> &b) {
  Integer<cont_t, alloc_t> result;
  typename Integer<cont_t, alloc_t>::iterator ia=a.begin(), ib=b.begin();
```

[3] We used composition in our implementation. The main reason was that we wanted to minimize the tradeoff between long numbers consisting of just one digit and real long numbers. Therefore, our `Integer` class is in fact a kind of union or variant record in Pascal notation of either a pointer to the parameterized container or a plain digit. The source code of our implementation is available at http://www-ca.informatik.uni-tuebingen.de/people/simonis/projects.htm.

```
  while(ia != a.end()) result.push_back(*ia + *ib);)
  return result;
}
```

Based on the technique of iterator traits described in [5] and the proposed container traits in [4] specialized versions of certain algorithms may be written, which make use of the specific features of the underlying container. For example, an algorithm working on vectors can take advantage of random access iterators, while at the same time being aware of the fact that insert operations are linear in the length of the container.

5 Conclusions and Perspectives

We have shown how template template parameters are typically employed. They can be used to give library and class designers new power in providing the user with a facility to adapt the predefined behavior of classes and functions according to his needs and application context. This is especially important if one wants to build on top of already existing generic libraries like the STL.

With our example we demonstrate how template template parameters and generic programming can be used to achieve a flexible design. In contrast to usual template parameters, which parameterize with concrete types, template template parameters allows one to parameterize with incomplete types. This is a kind of *structural* abstraction compared to the abstraction over simple types achieved with usual template parameters. As templates are always instantiated at compile time, this technique comes with absolutely no runtime overhead compared to versions which don't offer this type of parameterization.

One has to think about the applicability of template template parameters, a C++ feature, to other programming languages. Generally, a similar feature makes sense in every language that follows C++'s instantiation model of resolving all type bindings at compile time (e.g., Modula-3 and Ada). Template template parameters are a powerful feature to remove some restrictions imposed by such a strict instantiation model without introducing runtime overhead.

Table 1. Performance comparison of our Integer class compared to other arbitrary precision libraries. While GMP is a C library with optimized assembler routines, all the other libraries are written in C++. The first line of every entry denotes the number of processor instructions while the second one indicates the number of processor cycles needed for one operation. Integer† stands for `Integer<slist>`, Integer‡ for `Integer<std::list>`, and Integer§ for `Integer<std::vector>`. The "RW-" prefix marks tests, which have been taken with the Rogue Wave STL in contrast to the other tests, which used the SGI STL.

bits	GMP	CLN	NTL	Piologie	Integer†	Integer‡	RW-Integer‡	Integer§	RW-Integer§
Addition of n-Bit numbers									
128	820	718	2.087	509	3.693	4.119	3.275	1.658	2.038
	3.846	4.375	6.080	3.290	6.186	6.723	7.186	4.411	4.014
1.024	1.001	871	2.649	1.210	16.769	18.025	7.671	3.078	2.974
	4.336	4.596	6.350	5.392	17.512	17.961	10.892	5.574	4.497
8.192	1.691	1.971	7.350	3.748	121.805	128.774	40.046	13.456	11.668
	5.317	7.986	12.603	8.601	104.750	113.095	45.748	14.231	11.241
65.536	8.174	10.505	43.530	23.491	960.955	1.015.816	278890	96.657	80.150
	32.217	48.811	65.176	52.254	844.618	952.695	339371	104.143	77.745
Multiplication of n-Bit numbers									
128	922	990	1.900	1.293	6.181	6.687	4.088	2.798	2.646
	6.450	4.513	6.972	5.461	11.744	12.674	11.601	9.682	6.743
1.024	8.815	13.740	28.837	34.383	71.585	72.311	52.074	40.234	32.564
	16.572	24.736	29.671	43.539	60.661	63.933	48.949	35.594	30.221
8.192	240.738	394.093	828.825	940.873	2.852.491	2.786.261	2.749.538	2.399.391	1.882.072
	243.438	523.903	671.630	926.693	1.853.809	1.971.483	18.67.466	1.715.800	1.488.350
65.536	5.477.327	13.370.805	22.798.590	28.939.305	167.792.489	163.149.346	171.090.108	151.754.471	118.611.246
	5.158.666	14.695.137	18.031.103	27.289.599	117.485.455	122.771.953	120.008.350	107.798.237	93.442.537

We measured our example with GCC 2.97 and two versions of the STL, namely the from SGI [23] and one from Rogue Wave [20]. Table 1 compares our `Integer` class with some widely available arbitrary precision libraries (GMP 3.1.1 [11], CLN 1.0.3 [12], NTL 4.1a [22] and Piologie 1.2.1 [24]). The tests have been done on a PentiumIII 667MHz Linux system using the PCL library [6].

The results of some tests with garbage collected containers using the Boehm-Weiser-Demers [7] collector have been not very promising. However the significant performance difference between the two STL versions we used indicate that this may be no fundamental problem, but a problem of bad compiler optimization and the orthogonal design of the SGI-STL containers and the plain Boehm-Weiser-Demers garbage collector. Therefor we plan further tests in the future using optimizing compiler and other collectors like TGC [18], [19], which address exactly this problems.

6 Compiler Support

One major problem in working with template template parameters is not a conceptual, but rather a practical one. Even now, three years after the publication of the ISO C++ standard, not all compilers implement this feature.

We were able to compile our examples only with the following compilers: Borland C++ V5.5 [3], Visual Age C++ V4.0 [13], Metrowerks V6.0 and all compilers based on the edg front-end V2.43 [8]. The snapshot versions after November 2000 of the Gnu C++ Compiler [10] also meet the requirements.

7 Acknowledgements

We want to thank Rüdiger Loos for his ideas on nested lists and long integers based on STL containers, which sparked our interest in template template parameters. We also want to thank the guys from EDG [8] for always supplying us with the newest version of their fabulous compiler and Martin Sebor from Rogue Wave for making available to us the latest version of their STL.

References

1. Andrei Alexandrescu. *Modern C++ Design*, Addison-Wesley Publishing Company, 2001.
2. ANSI/ISO Standard. *Programming languages — C++*, ISO/IEC 14882, 1998.
3. Borland, Inprise Corporation. *Borland C++ Compiler 5.5*, www.borland.com/bcppbuilder/freecompiler.
4. Matthew Austern. *Algorithms and Containers*, C++ Report, p. 44–47, 101 communications, July/August 2000.
5. Matthew Austern. *Generic Programming and the STL*, Addison-Wesley Publishing Company, 1999.
6. Rudolf Berrendorf, Bernd Mohr. *PCL — The Performance Counter Library*, www.fz-juelich.de/zam/PCL.
7. Hans Boehm. *Boehm-Weiser-Demers collector*, www.hpl.hp.com/personal/Hans_Boehm/gc.
8. Edison Design Group. *The C++ Front End*, www.edg.com/cpp.html.
9. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns*, Addison-Wesley Publishing Company, 1995.
10. GCC steering committee. *GCC Home Page*, gcc.gnu.org.
11. Torbjorn Granlund. *GNU MP — The GNU Multi Precision Library*, www.swox.com/gmp.
12. Bruno Haible. *CLN*, clisp.cons.org/~haible/packages-cln.html.
13. IBM. *IBM VisualAge C++*, www.software.ibm.com/ad/visualage_c++.
14. Mehdi Jazayeri, Rüdiger G. K. Loos, David R. Musser (Eds.). *Generic Programming*, LNCS No. 1766, Springer, 2000.
15. David R. Musser, Gillmer J. Derge, Atul Saini. *STL Tutorial and Reference Guide: Second Edition*, Addison-Wesley Publishing Company, 2001.
16. David R. Musser, S. Schupp, Rüdiger Loos. *Requirement Oriented Programming*, in [14], p. 12-24, 2000.
17. Scott Meyers. *Effective C++ CD*, Addison-Wesley Publishing Company, 1999.
18. Gor V. Nishanov, Sibylle Schupp. *Garbage Collection in Generic Libraries*, Proc. of the ISMM 1998, p. 86-97, Richard Jones (editor), 1998.
19. Gor V. Nishanov, Sibylle Schupp. *Design and implementation of the fgc garbage collector*, Technical Report98-7, RPI, Troy, 1998.
20. Rogue Wave Software. *Rogue Wave STL (development snapshot)*, November 2000.
21. Volker Simonis, Roland Weiss. *Heterogeneous, Nested STL Containers in C++*, LNCS No. 1755 (PSI '99): p. 263-267, Springer, 1999.
22. Victor Shoup. *NTL*, www.shoup.net/ntl.
23. STL Team at SGI. *Standard Template Library Programmer's Guide*, www.sgi.com/Technology/STL.
24. Sebastian Wedeniwski. *Piologie*, ftp://ftp.informatik.uni-tuebingen.de/pub/CA/software/Piologie.

Compiler-Cooperative Memory Management in Java

Vitaly V. Mikheev, Stanislav A. Fedoseev

A. P. Ershov Institute of Informatics Systems,
Excelsior, LLC
Novosibirsk, Russia
e-mail: {vmikheev,sfedoseev}@excelsior-usa.com

Abstract. Dynamic memory management is a known performance bottleneck of Java applications. The problem arises out of the Java memory model in which all objects (non-primitive type instances) are allocated on the heap and reclaimed by garbage collector when they are no longer needed. This paper presents a simple and fast algorithm for inference of object lifetimes. Given the analysis results, a Java compiler is able to generate faster code, reducing the perfromance overhead. Besides, the obtained information may be then used by garbage collector to perform more effective resource clean-up. Thus, we consider this technique as "compile-time garbage collection" in Java.

Keywords: Java, escape analysis, garbage collection, finalization, performance

1 Introduction

Java and other object-oriented programming languages with garbage collection are widely recognized as a mainstream in the modern programming world. They allow programmers to embody problem domain concepts in a natural coding manner without paying attention to low-level implementaion details. The other side of the coin is often a poor performance of applications written in the languages. The problem has challenged compiler and run-time environment designers to propose more effective architectural decisions to reach an acceptable performance level.

A known disadvantage of Java applications is exhaustive dynamic memory consumption. For the lack of stack objects — class instances put on the stack frame, all objects have to be allocated on the heap by the *new* operator. Presence of object-oriented class libraries makes the situation much worse because any service provided by some class, prerequires the respective object allocation. Another problem inherent to Java is a so-called *pending object reclamation* [1] that does not allow garbage collector to immediately utilize some objects even though they were detected as unreachable and finalized. The Java Language Specification imposes the restriction on an implementation due to the latent caveat: if an object has a non-trivial finalizer (the **Object.finalize()** method overriden) to perform some post-mortem clean-up, the finalizer can resurrect its object "from the dead", just storing it, for instance, to a static field. Pending object reclamation reduces memory resources available to a running application.

Generally, performance issues can be addressed in either compiler or run-time environment. Most Java implementations (e.g. [2] [3]) tend to improve memory management by implementing more sophisticated algorithms for garbage collection [4]. We strongly believe that the mentioned problems should be covered in both compile-time analysis and garbage collection to use all possible opportunities for performance enhancement.

Proposition 1. *Not to junk too much is better than to permanently collect garbage*

We propose a scalable algorithm for object lifetime analysis that can be used in production compilers. We implemented the system in **JET**, Java to native code compiler and run-time environment based on the Excelsior's compiler construction framework [5].

The rest of the paper is organized as follows: Section 2 describes the program analysis and transformation for allocating objects on the stack rather than on the heap, Section 3 describes our improvements of the Java finalization mechanism. The obtained results are presented in Section 4, Section 5 highlights related works and, finally, Section 6 summarizes the paper.

2 Stack Allocating Objects

In Java programs, the lifetimes of some objects are often obvious whereas the lifetimes of others are more uncertain. Consider a simple method, getting the current date:

```
int foo {
  Date d  = new Date();
  return d.getDate();
}
```

At the first glance, the lifetime of the object *d* is resctricted to that of method *foo*'s stack frame. That is an opportunity for a compiler to replace the *new* operator with a stack allocating object. However, we have to guarantee that no *d* aliases *escape* from the stack frame, that is, no aliased references to *d* are stored anywhere else. Otherwise, such program transformation would not preserve the original Java semantics. In the above example, the method *getDate* is a possible "escape direction".

Escape analysis dating back to the middle 1970s [6], addresses the problem. Many algorithms proposed vary in their application domains and time and spatial complexity. We desighed a simple and fast version of escape analysis specially adapted to Java. Despite its simplicity, the algorithm shows promising results of benchmarking against widespread Java applications.

2.1 Definitions

All variables and formal parameters in the below definitions are supposed to be of Java reference types. By definition, formal parameters of a method also include the implicit "this" parameter (method receiver).

Definition 1 (Alias). *An expression expr is an alias of a variable v at a particular execution point, if v == expr (both v and expr refer to the same Java object)*

Definition 2 (Safe method). *A method is safe w.r.t its formal parameter, if any call to the method does not create new aliases for the parameter except, may be, a return value*

Definition 3 (Safe variable). *A local frame variable is safe, if no its aliases are available after method exit*

Definition 4 (Stackable type). *A reference type is stackable, if it has only a trivial finalizer*

Definition 5 (A-stackable variable). *A safe variable v is A-stackable, if a definition of v has the form of v = new T() for some stackable type*

Definition 6 (Stackable variable). *An A-stackable variable is stackable, if no local aliases of the variable exist before a repetitive execution of the variable definition in a loop, if any*

The stackable type definition is used to hinder a possible reference escape during finalizer invocation. The A-stackable to stackable variable refinement is introduced to preserve the semantics of the *new* operator: being executed in a loop, it creates different class instances so the analysis has to guarantee that previously cretead instances are unavailable.

2.2 Program Analysis and Transformation

To detect if a variable is not safe, we distinguish two cases of escape:

1. *explicit*: **return v**, **throw v** or **w.field = v** (an assignment to a static or instance field)
2. *implicit*: **foo (...,v, ..)** invocation of a method non-safe w.r.t **v**

Operators like **v = v1** are subject for a flow-insensitive analysis of local reference aliases (LRA) [10]. In order to meet the requirement for loop-carried variable definitions, the algorithm performs a separate LRA-analysis within loop body. Determining of safe methods is proceeded recursively as a detection of their formal parameter safety except the *return* operator. In such case, the return argument becomes involved into local reference aliasing of the calling method. We implemented our algorithm as a backward inter-procedural static analysis on call graph like algorithms described in related works [7],[9]. We omit the common analysis scheme due to its similarity to those of related works and focus on some important differencies further.

Once stackable variables have been detected, the respective *v* = *new T()* operators are replaced with the *v* = *stacknew T()* ones from internal program representation. Besides, the operators like *v* = *new T[expr]*, allocating variable length arrays are marked with a tag provided for subseqent code generation. That makes sense because our compiler is able to produce code for run-time stack allocation.

2.3 Implementation Notes

The Excelsior's compiler construction framework features a statistics back-end component [5] making it a suitable tool of statistic gathering and processing for any supported input language. Also, we had a memory allocation profiler in the run-time component so we were able to analyze a number of Java applications. We found that the algorithms described in related works may be somewhat simplified without sacrificing effectiveness. Moreover, the simplification often leads to better charateristics such as compilation time and resulting code size.

Type inference. So far, we (implicitly) supposed that all called methods are available for analysis. However, Java being an object-oriented language, supports virtual method invocation — run-time method dispatching via Virtual Method Tables that hinders any static analysis. Type inference [13] is often used to avoid the problem to some extent. Our algorithm employs a context-sensitive local type inference: it starts from the known local types sourcing from local *new T()* operators and propagates the type information to called method context. We used a modified version of the rapid type inference pursued in [12]. Another opportunity which helps to bypass the virtual method problem is global type inference based on the class hierarchy analysis [11]. We implemented a similar algorithm but its applicability is often restricted because of the Java dynamic class loading. We did not consider polyvariant type inference (analysis of different branches at polymorphic call sites) due to its little profit in exchange for the exponential complexity.

Inline substitution. Local analysis in optimizing compilers is traditionally stronger than inter-procedural because, as a rule, it requires less resources. This is why inline substitution not only removes call overhead but also often improves code optimization. Escape analysis is not an exception from the rule: local variables that were not stackable in the called method may become so in the calling one, for instance, if references to them escaped via the *return* operator. Escape analysis in Marmot [7] specially treats called methods having that property to allocate stack variables on the frame of calling method. In the case, called method should be duplicated and specialized to add an extra reference parameter (Java supports metaprogramming so the original method signature may not be changed). In our opinion, that complicates analysis with no profit: the same problem may be solved by an ordinary inline substitution without the unnecessary code growth.

Native method models. The Java language supports external functions called *native methods*. They are usually written in C and unavailable for static analysis. However, certain native methods are provided in standard Java classes and should be implemented in any Java run-time or even compiler, for instance the **System.arraycopy** method. Because the behaviour of such methods is strictly defined by the Java Language Specification [1], we benefit from using so-called *model methods* provided for analysis purposes only. A model native method has a fake implementation simulating the original behaviour interesting for analysis. Employing model methods improves the overall precision of escape analysis.

2.4 Complexity

In according to [14], given restrictions even weaker than ours, escape analysis can be solved in linear time. The rejecton of analyzing polyvariant cases at virtual call sites and the restriction of reference aliasing to local scopes only give the complexity proportional to N (program size) + G (non-virtual call graph size). Thus, our algorithm performs in O(N+G) both time and space.

3 Finalization

The described algorithm determining *safe methods* may be used for more effective implementation of pending object reclamation in Java. As mentioned above, an object having a non-trivial finalizer is prevented from immediate discarding by a garbage collector. The main problem provoking a significant memory overhead is that all heap subgraph reachable from the object may not be reclaimed as well: finalizer may potentially "save" (via aliasing) any object from the subgraph.

To overcome the drawback, we adapted the algorithm to detect whether the finalizer is a safe method with respect to its implicit "this" parameter and other object's fields aliased from "this". The analysis results are then stored by compiler to the class object (a Java metatype instance [1]). Given that, garbage collector makes a special treatment for objects with trivial or safe finalizers. More specifically, the run-time system constructs a separate list for objects which require pending reclamation whereas other objects are processed in a simpler way. The measurement results for the optimization are listed in the next section.

4 Results

We implemented the described optimizations as a part of the JET compiler and run-time environment. We selected the Javacc parser generator, the Javac bytecode compiler from Sun SDK 1.3 and Caffein Dhrystone/Strings benchmarks to evaluate resulting performance of the escape analysis application. The results are shown in Table 1 (the numbers were computed as *NewExecutionTime/OldExecutionTime*). The performance growth is achieved as a result of both faster object allocation and less extensive garbage collection.

These tests were choosen due to their batch nature that allows us to measure the difference in total execution time. Despite the results for the first three benchmarks are valuable, applying the optimization to the Javac compiler had only minimal effect — no silver bullet. Unfortunately, the results may not be directly compared with the results obtained by other researchers. The comparison of different algorithms may be accomplished only within the same optimization and run-time framework. For instance, a system with slower object allocation and garbage collection or better code optimization would obviously experience more significant performance improvement from the stack allocation.

Results of optimized finalization are given in Table 2. JFC samples (Rotator3D, Clipping, Transform, Lines) using Java 2D-graphics packages were chosen because of very intensive memory consumption. We measured the amount of free memory just after garbage collecting and the numbers were computed as *NewFreeMemory/OldFreeMemory*. The total amount of heap memory was the same for all tests and equal to 30MB.

Table 1. Stack allocating objects

Benchmark	Execution time fraction
Javacc	0.54
Dhrystone	0.32
Strings	0.2
Javac	0.98

Table 2. Optimized finalization

Benchmark	Free memory fraction	Memory profit, MB
Rotator3D	1.1	+1.5
Clipping	1.15	+1.2
Transform	1.08	+0.7
Lines	1.13	+1.7

We noted that even with the optimizations enabled, the total compilation time remains virtually unchanged. Analyzing obtained results, we draw a conclusion that the considered object-oriented optimizations may be employed by production compilers. All further information related to the JET project may be found at [18].

5 Related Works

An number of approaches have been proposed for object lifetime analysis. Many works were dedicated to functional languages such as SML, Lisp etc. ([14], [15], [16]). The power of the escape analyses supercedes ours to a great extent, however the complexity of the algorithms is not better than polynomial. The escape analysis for Java was investigated by reseachers using static Java analyzing frameworks. Except the JET compiler, the related works were completed on the base of the TurboJ via-C translator [9], the IBM HPJ compiler [10] and the Marmot compiler project at Microsoft Research [7]. The algorithm presented is simpler but, nevertheless, quite effective and precise so it may be used even in dynamic compilers built in the most current Java Virtual Machines [2], [3]. Besides, the related works discuss only stack allocating objects whereas our approach also considers garbage collection improvement basing on the compile-time analysis.

6 Conclusion

This paper presented a technique for fast and scalable object lifetime analysis. Being used in cooperative compiler and run-time framework, the implemented optimizations profit in both execution speed and memory consumption of Java applications. The interesting area for future works is to investigate a region inference algorithms allowing compiler to approximate object lifetimes between method call boundaries. Despite the applicability of such analysis to compiler optimizations is doubt, the information may be used for more effective garbage collection in compiler-cooperative run-time environment.

References

1. J. Gosling, B. Joy and G.Steele: *The Java Language Specification.* Addison-Wesley, Reading, 1996
2. Gu et al.: *The Evolution of a High-Performing JVM.* IBM Systems Journal, Vol. 39, No. 1, 2000
3. *The Java HotSpot(tm) Virtual Machine*, Technical Whitepaper, Sun Microsystems Inc., Whitepaper, 2001. **http://www.sun.com/solaris/java/wp-hotspot**
4. D. Detlefs, T. Printezis: *A Generational Mostly-Concurrent Garbage Collector.* In Proc. ACM ISMM'00, 2000
5. V.V. Mikheev: *Design of Multilingual Retargetable Compilers: Experience of the XDS Framework Evolution* In Proc. Joint Modular Languages Conference, JMLC'2000, Volume 1897 of LNCS, Springer-Verlag, 2000
6. J.T. Shwartz: *Optimization of very high-level languages -I. Value transmission and its coloraries.* Computer Languages, 1(2), 1975
7. D. Gay, B. Steensgaard: *Stack Allocating Objects in Java.* Reasearch Report, Microsoft Research, 1999
8. J.-D. Choi et al.: *Escape Analysis for Java.* SIGPLAN Notices, Volume 34, Number 10, 1999
9. B. Blanchet: *Escape Analysis for Object Oriented Languages. Application to Java(tm).* SIGPLAN Notices, Volume 34, Number 10, 1999
10. D. Chase, M. Wegman, and F. Zadeck: *Analysis of pointers and structures.* SIGPLAN Notices, Volume 25, Number 6, 1990
11. D. Bacon: *Fast and Effective Optimization of Statically Typed Object-Oriented Languages.* Technical Report, University of California, 1997
12. D. Bacon, P. Sweeney: *Fast static analysis of C++ virtual calls.* In Proc. OOPSLA'96, SIGPLAN Notices, Volume 31, Number 10, 1996
13. O. Agesen: *The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism.* In Proc. ECOOP'95, Aarhus, Denmark, 1995
14. A. Deutch: *On the complexity of escape analysis.* In Conference Record of POPL'97, The 24th ACM SIGPLAN-SIGAST, 1997
15. S. Hughes: *Compile-time garbage collection for higher-order functional languages.* Journal of Logic and Computation, 2(4), 1992
16. K. Inoue et al.: *Analysis of functional programs to detect run-time garbage cells.* ACM Transactions on Programming Languages and Systems, 10(4), 1988
17. Y. Park, B. Goldberg: *Escape analysis on lists* In Proceedings of PLDI'92, ACM SIGPLAN, 1992
18. *JET Deployment Environment.* Excelsior LLC, Technical Whitepaper, 2001. **http://www.excelsior-usa.com/jetwp.html**

A Software Composition Language and Its Implementation

Dietrich Birngruber

Johannes Kepler University Linz,
Institute for Practical Computer Science
System Software Group
A-4040, Linz, Austria
e-mail: birngruber@ssw.uni-linz.ac.at

Introduction

Binary software components offer solutions to various software engineering problems, e.g. how to build and maintain complex software systems in a changing environment. The idea is to acquire prefabricated, well-tested and platform independent binary software components on the market and to compose them to new applications by plugging them together in builder tools without the need for coding. There are already markets [1] for components as well as some common understanding about the term software component [2].

The composition of binary software components divides the development process into two parts. First, component developers write new component libraries and second, application programmers use them to compose their applications. Often different individuals assume these roles. This leads to a knowledge gap, as the application programmer has to determine how and in which context he can apply the different components. Of course, a component provider has to state the component's context dependencies clearly in a proper documentation.

The full paper introduces the idea of component plans and their description in Component Plan Language (CoPL) that tries to bridge this gap and the idea of a component technology independent composition language as an XML [3] application, called Component Markup Language (CoML). Whereas in the paper we focus on the description of CoML.

CoPL — Component Plan Language

A component plan describes how an application programmer typically glues components of a given library together. A plan is a description of a composition with *Decision Spots*. Typically a plan is written in CoPL and captures domain knowledge and typical usage scenarios or composition patterns by providing a typical pre-wiring of the used components. The application programmer processes these CoPL plans with a generator. The generator produces CoML code which can be used by different IDEs for different component technologies. Figure 1 shows a typical usage scenario for CoPL and CoML.

The generator uses the plan as input and — if stated in the plan — asks the application programmer to substitute place-holders by concrete components from a list of matching component implementations. We call these place-holders *Decision Spots*. Currently the matching algorithm is based on type substitutability.

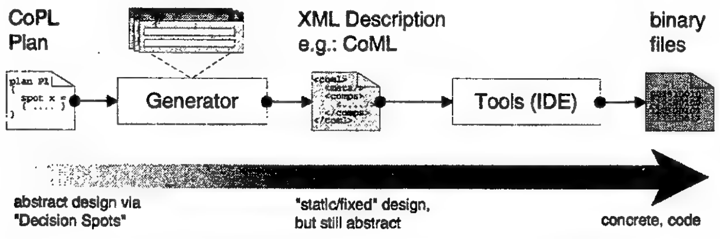


Fig. 1. CoPL and CoML Usage Scenario

On the one hand writing glue code manually gives the application assembler great flexibility, where on the other hand tools (e.g. wizards) automate routine and clearly predefined composition tasks, like generating a code snippet for a new GUI dialog. A possible way for combining the advantages of these composition techniques is to introduce a "script-able generator". In fact, a CoPL plan is used for scripting the generator. The interpreted plan guides application programmers semi-automatically (similarly to a wizard) through the assembly process for example by displaying a dialog for choosing the desired implementation (e.g. ArrayListImpl) for a given interface (e.g. for an "IList" interface). In contrast to a wizard, a plan is not fixed but can be modified. It is like a composition template with some degrees of freedom. A plan may contain Decision Spots that offer choices to the application programmer.

Considering a library with many components, it is a tedious task to find the right components and to instantiate and glue them together according to the desired composition pattern. Our component plans along with the generator automate this process by supplying the programmer with knowledge about how the component developer intended to wire the components.

CoPL is based on previous work on JavaBeans [4] composition using plans (see [5]). However, CoPL and CoML are not tuned toward a special component technology like JavaBeans, or Microsoft's .NET [6] components.

CoML — Component Markup Language

The Component Markup Language (CoML) is an XML application for composing software components. The main goal for CoML is to have a platform independent description of component composition which is processable by various software tools like development tools. CoML can be interpreted like other scripting languages. In our intention CoML should primarily be created and used by software tools rather than require human beings to manually write (and execute) CoML scripts. However, in the spirit of XML, we still tried to make CoML human readable as well and developed an interpreter for the Java and the .NET platform.

In order to keep CoML component model independent we had to define minimum requirements for component models script-able by CoML:

- a component is strongly typed
- a component is accessed via interface(s)
- a component interface offers methods and/or properties
- a component uses an event-mechanism as its primary "wiring" technique for plugging components together
- the component life-cycle is split into design-time and run-time and thus between wiring components versus creating instances

Based on this assumptions about component models, CoML offers first class abstractions (i.e. XML tags) for describing a component composition. CoML tags can be used for:

- defining the component itself
- setting and getting properties
- defining event-bindings and thus wireing differnt components
- placing method-calls
- building up a containment hierarchy

Example 1. This CoML snippet shows a simple composition of two GUI components — a slider and a progress bar. When the slider is moved the slider's current value is displayed in the progress bar. Slider and progress bar are connected via the `change` event. The slider is the event source and the method **`setValue`** of the progress bar the event sink. The target component platform is JavaBeans from Sun.

```
<component id="progBar1" class="javax.swing.JProgressBar">
  <property name="value" access="set">
    <int>50</int>
  </property>
</component>
<!-- progress bar reacts upon the sliders change event -->
<component id="slider" interface="ISlider" class="ssw.webui.Slider">
  <on-event name="change" filter="stateChanged">
    <sink-method name="setValue" idRef="progBar1">
      <property idRef="slider" name="value" access="get"/>
    </sink-method>
  </on-event>
</component>
```

Related Work

Sun and IBM have developed their own composition language based on the meta syntax XML. However both, Sun's JavaBean Persistence [8] and IBM's Bean Markup Language (BML) [9], are tailored for JavaBeans.

The main goal of Sun's approach is to have a proprietary standardized format for exchanging mainly GUI JavaBeans compositions between different Java IDEs. At the beginning of the project we tried to use Bean Persistence as the primary output of the Generator. Unfortunately Bean Persistence expressiveness for composing components via events is too limited for our purposes.

CoML is influenced by BML. The main differences are that CoML is not focused on JavaBean composition, that CoML supports interfaces where BML allows explicit type conversions, that CoML does not allow embedding of foreign scripting code - like JavaScript [10] — in order to remain platform independent.

Conclusions

An application programmer uses component plans at design-time, i.e. when he assemblies the components to a new application. The benefits are to have a script-able wizard, that produces a platform independent description of a concrete component composition.

Plans along with the generator are used at a different point in time than scripting languages, which are typically interpreted or executed (like e.g. Piccola [7], JavaScript or IBM's Bean Markup Language). These languages are interpreted at run-time, i.e. at the end user's computer during actual execution of the application.

The output of the generator is a composition description in CoML. CoML is component technology and platform independent. Different tools like development tools, documentation tools or software architecture visualizing tools can use CoML for e.g. exchanging component compositions or displaying them in different manners. Of course, CoML can be interpreted as well and currently we have interpreters for Java and Microsoft's .NET component platform. We have a research prototype for composing JavaBeans which uses CoML as its persistence format.

References

[1] http://www.developer.com/directories
[2] Szyperski Clemens: Component Software — Beyond Object-Oriented Programming. Addison-Wesley. 1997.
[3] W3C: Extensible Markup Language (XML) 1.0. W3C Recommendation. 10. Feb. 1998. http://www.w3.org/TR/REC-xml.
[4] Sun Microsystems: JavaBeans API Specification. 1997. http://java.sun.com/beans/spec.html.
[5] Birngruber Dietrich, Hof Markus: Using Plans for Specifying Preconfigured Bean Sets. In: Li Qiaoyun, et. al. (Eds.): Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 34). Santa Barbara, CA. 2000.
[6] Richter Jeffrey: .NET Framework: Building, Packaging, Deploying, and Administering Applications and Types. in: MSDN magazine. February 2001.
[7] Achermann Franz, Nierstrasz Oscar: Applications = Components + Scripts – A tour of Piccola. in: Mehmet Aksit (Ed.): Software Architectures and Component Technology. Kluwer. 2000.
[8] Milne Philip, Walrath Kathy: Long-Term Persistence for JavaBeans. Sun Microsystems. 24. Nov. 1999.
[9] Curbera Francisco, Weerawarana Sanjiva, Duftler Matthew J.: On Component Composition Languages. in: Bosch, Szyperski, Weck (Ed.): Proceedings of the Fifth International Workshop on Component-Oriented Programming (WCOP 2000). 2000. ISSN 1103-1581. see also http://www.alphaWorks.ibm.com/formula/bml.
[10] ECMA 262: ECMAScript Language Specification. 3rd Edition. December 1999. http://www.ecma.ch.

Editor Definition Language and Its Implementation

Audris Kalnins, Karlis Podnieks, Andris Zarins, Edgars Celms, Janis Barzdins

Institute of Mathematics and Computer Science,
University of Latvia
Raina bulv. 29, LV-1459, Riga, Latvia
{audris, podnieks, azarins, edgarsc, jbarzdin}@mii.lu.lv

Abstract. Universal graphical editor definition language based on logical metamodel extended by presentation classes is proposed. Implementation principles based on Graphical Diagramming Engine are also described.

1 Introduction

Universal programming languages currently have become more or less stable. However, the development of specialised programming languages for specific areas is still going on (most frequently, this type of languages is no more called programming languages, but specification or definition languages). One of such specific areas is the definition of graphical editors. In this paper the **Editor Definition Language (EdDL)** for a simple and convenient definition of wide spectrum of graphical editors is proposed, and the basic implementation principles of EdDL are described.

Let us mention some earlier research in this area. Perhaps, the first similar approach has been by Metaedit [1], but its editor definition facilities are fairly limited. The most flexible definition facilities seem to be the Toolbuilder by Lincoln Software. Being a typical meta-CASE of early nineties, the approach is based on an extended ER model for describing the repository contents and for defining derived data objects which are in one-to-one relation to objects in a graphical diagram. A more academic approach is that proposed by Kogge [2], with a very flexible, but very complicated procedural editor definition language. Another similar approaches are proposed by DOME [7] and Moses [8] projects, with fairly limited definition languages. Several commercial modelling tools (STP by Aonix, ARIS by IDS prof. Scheer etc) use a similar approach internally, for easy customisation of their products.

Our approach in a sense is a further development of the above-mentioned approaches. We develop the customisation language into a relatively independent editor definition language (EdDL), which, on the other hand, is sufficiently rich and easy to use, and, on the other hand, is sufficiently easy to understand. At the same time it can be implemented efficiently, by means of the universal Editor Engine. Partly the described approach has been developed within the EU ESPRIT project ADDE [3], see [4] for a preliminary report.

2 Editor Definition Language. Basic Ideas

The proposed editor definition language consists of two parts:

- the language for defining the logical structure of objects which are to be represented graphically
- the language for defining the concrete graphical representation of the selected logical structure.

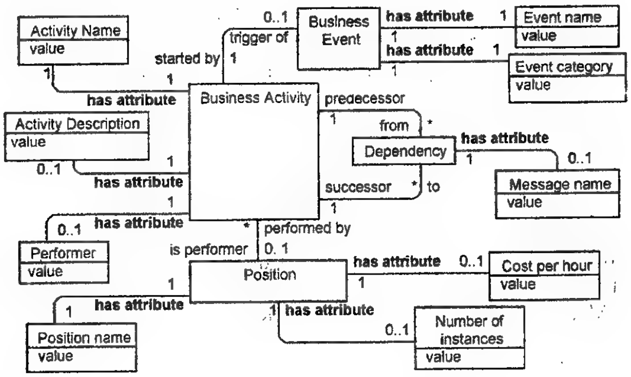


Fig. 1. Logical metamodel example

For describing the logical structure there exists a generally adapted notation — UML class diagrams [5], which typically is called the **logical metamodel**. Fig.1 shows a simple example of a logical metamodel for business activities domain.

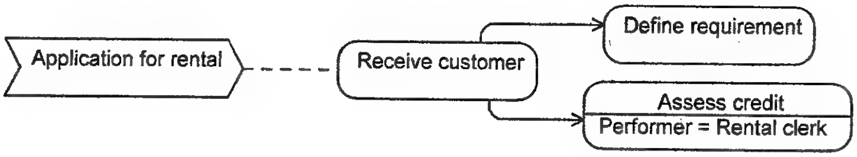


Fig. 2. Business activity diagram example

The **EdDL** language will be presented as an extension of the logical metamodel. Let us assume that we want to present the Business activity domain by diagrams similar to that depicted in fig.2. Fig.3 demonstrates the use of EdDL for the definition of the example editor (with some minor details omitted). In this figure rectangles represent the same classes from the logical metamodel in fig. 1, but rounded rectangles represent classes being the proper elements of EdDL. The first element added to the logical metamodel is the **diagram** class (*Business activity diagram*), together with standard associations (with the role name **contains**). One more standard association for the diagram is the refinement association (**refines**), which defines that a *Business Activity* can be further refined by its own *Business activity diagram.*

Each of the metamodel classes, which must appear as graphical objects in the diagram, are linked by an unnamed association to its **presentation class** — a subclass of standard classes **box** or **line**. For example, the presentation class for *Business Activity* – the *Activity box* class says that every business activity must be represented by a rounded rectangle in a light blue default colour. The *Icon* representing this graphical symbol on the editor's **symbol palette** is also shown. Lines are presented in a similar way, for showing their direction the relevant role names from the metamodel are referenced in the presentation class (e.g. **start**=*predecessor*).

The most interesting element in this EdDL example is the definition of **prompting** and **navigation**. Prompting here means the traditional service found in an editor that a value can be selected from the offered list of values (value of *Performer* selected from the list of available *Position names*). The navigation means the editor feature that double-clicking on the *Performer* field in a box automatically invokes some default editor for *Position.* Both *Prompting* and *Navigation* are shown in the EdDL as fixed classes linking the attribute to the relevant association.

3 EdDL Implementation Principles

EdDL has been implemented by means of an interpreter, which in this case is named **Editor Engine**. When an editor has been defined in EdDL the Editor Engine acts as the desired graphical editor for the end user. A key aspect is that Editor Engine (EE) relies on **Graphical Diagramming Engine (GDE)** for all diagram drawing related activities. The primitives implemented by GDE — diagram, box, line, compartment etc. and the supported operations on them are very fit for this framework. Thus the interface between EE and GDE is based on very appropriate high level building blocks. The GDE itself was developed by IMCS UL initially within the framework of ADDE project, with a commercial version later on. It is based on very sophisticated graph drawing algorithms [6].

The practical experiments on using EdDL and EE have confirmed the efficiency and flexibility of approach. The defined editors behave as industrial quality graphical editors. The flexibility has been tested by implementing full UML 1.3 and various extensions to it for modelling business processes.

References

1. Smolander, K., Martiin, P., Lyytinen, K., Tahvanainen, V-P.: Metaedit — a flexible graphical environment for methodology modelling. Springer-Verlag (1991)
2. Ebert, J., Suttenbach, R., Uhe, I.: Meta-CASE in Practice: a Case for KOGGE. Proceedings of the 9th International Conference, CAiSE'97. Barcelona, Catalonia, Spain (1997)
3. ESPRIT project ADDE. http://www.fast.de/ADDE
4. Sarkans, U., Barzdins, J., Kalnins, A., Podnieks, K.: Towards a Metamodel-Based Universal Graphical Editor. Proc. of the Third International Baltic Workshop DB&IS, Riga, (1998)
5. Rumbaugh, J., Booch, G., Jackobson, I.: The Unified Modeling Language Reference Manual. Addison-Wesley (1999)

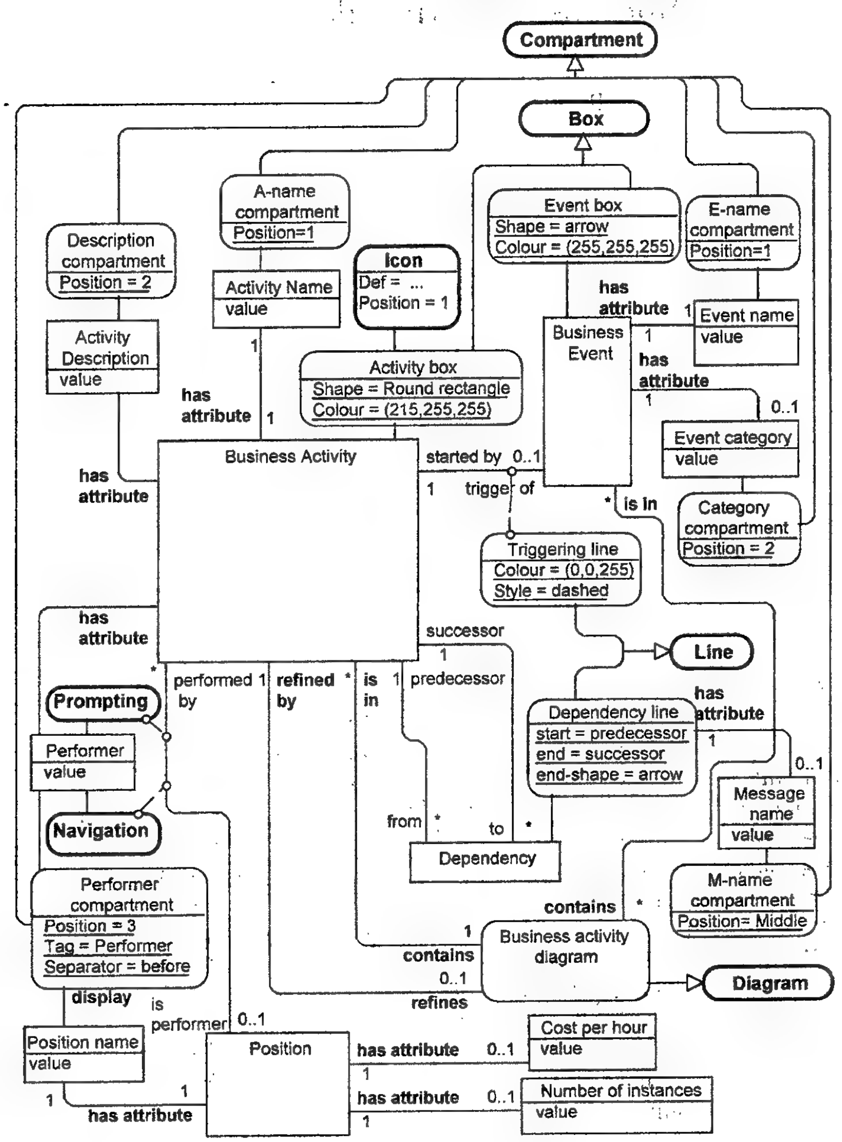


Fig.3 Business activity diagram editor definition in EdDL

6. Kikusts, P., Rucevskis, P.: Layout Algorithms of Graph-like Diagrams for GRADE-Windows Graphical Editors. Lecture Notes in Computer science. Vol. 1027. Springer-Verlag (1996) 361—364
7. DOME Users Guide. http://www.htc.honeywell.com/dome/support.htm
8. Esser, R.: The Moses Project. http://www.tik.ee.ethz.ch/~moses/MiniConference2000/pdf/Overview.PDF

Oberon-2 as Successor of Modula-2 in Simulation

Alexey S. Rodionov[1] and Dmitry V. Leskov[2]

[1] Institute of Computational Mathematics and Mathematical Geophysics SB RAS
Novosibirsk, RUSSIA
phone: +383-2-396211, email: alrod@rav.sscc.ru
[2] A. P. Ershov Institute of Informatics Systems,
Siberian Division of the Russian Academy of Sciences
6, Acad. Lavrentiev ave., 630090, Novosibirsk, Russia

Abstract. In the report the programming languages Oberon-2 is discussed from the point of view of convenience to program the discrete event simulation systems (DESS). Its predecessor Modula-2 was used as the basic language for a number of simulation packages and was proved to be good for it, but has Oberon-2 enough features to replace it and stand against domination of C++ in this special area? Are there compilers and programming environments good enough for this purpose? Is it possible to use existent libraries and transfer software between different platforms? These and other questions are discussed on the examples of ObSim-2 simulation package and XDS Modula-2&Oberon-2 programming system.

1 Introduction

The programming language Modula-2 for a long time has attracted attention of the developers of DESS. It was proved to be a convenient tool for the development of well-structured programs with the possibility of organization of quasi-parallel processes. The later is of a special importance for Simula-like DESS. A number of simulation packages on Modula-2 were designed [1]-[7]. Moreover, Modula-2 was proved to be so good programming language for simulation packages that it was used as a basis for the design of special simulation languages. For example, one the most powerful simulation systems of the late, MODSIM III [12], has Modula-like language.

One of the authors designed the package SIDM-2 [8] using Modula-2 as the basic language.

Other author is one of the designers of the XDS Modula-2 and Oberon-2 programming system. The XDS Modula-2 and Oberon-2 programming system allows, at first, to use the object-oriented resources of the second language, and secondly to transfer to the new operational environment earlier programmed non-object-oriented part of the package SIDM-2. Doing it is possible completely to adhere to the standard ISO, that makes possible the creation of the really portable simulation package. This new package ObSim-2 includes some modules on the language Modula-2, providing generation of the pseudo-random values and processes, matrix calculations and data processing. The Oberon-2 modules are intended for the description of frame and behavior of simulated systems. The compatibility of XDS multi-language programming system with main C++ compilers allows using a number of useful libraries also.

2 Advantages and Shortcomings of Modula-2 as Programming Language for DESS

In [2] Modula-2 was discussed as a basic language for the DESS design. It is well known [9] that any simulation language ought to provide the following features:

- means for the data organizing that provide simple and effective simulation;
- convenient means for the formulation and running the dynamic properties of a simulated system;
- possibility to simulate stochastic systems, i.e. procedures for generating and analysis of random variables and time series.

Now we can add that the object orientation is also of prime importance. Really, it could be said that OOP originated from simulation (refer to Simula-67! [11]).

It is clear to see that Modula-2 satisfies all demands but one: it has no standard modules for statistical support of a simulation process (no pseudo-random generators and data processing), but this is not a serious shortcoming as it is not very hard to design a special module with appropriate procedures. As for object-oriented properties, Modula-2 is an intermediate language. Modular concepts of this language allow to interpret some object-oriented features. Some extensions (TopSpeed for example []) include real class specifications. Moreover, Modula-2 has such valuable feature as quasi-parallel programming, that makes possible (with restrictions) process-oriented simulation. That explains why it was popular for DESS design in the late 1980th and early 1990th. Simulation package SIDM-2 also was programmed on Modula-2 because of its good convenience for the purpose.

This package provides the description of systems in the terms of discrete interactive processes (as in Simula) and events (similar to Simscript). This experience proves that Modula-2 is good for the purpose, but the TopSpeed extensions, first of all the object-oriented extension (classes) of the language were essentially used for rises of the efficiency of programs. The last circumstance has made the package hardly portable. At the same time SIDM-2 clearly shows that object-oriented features are of the prime importance for the efficiency and usability of simulation tools.

Processes are the part of Modula-2 that makes it good for the design of process-oriented DESS (Simula-like), but the process concept in Modula-2 has one severe shortcoming: it is simple to create any number of processes dynamically, but it is impossible to remove one from the program. According to the language description *end of any process is the and of a whole program.*

Strict typing is one of the main advantages of Modula-2. It allows to avoid a lot of possible mistakes on the early stages of the program model development. At the same time *general pointer (type ADDRESS)* allows to create indirect transition of parameters to event procedures and processes. Using that is dangerous but effective. Thus, in SIDM-2 event procedures have the following type:

```
TYPE EVENT_PROC = PROCEDURE(ParField : ADDRESS);
```

When one ties event with procedure he use the special procedure `Event^.SetProc` while to designate the parameters one ought to create an example of structure designed for this special kind of event and then ties it with event procedure using another special procedure `Event^.SetPars`. Let us to illustrate this by the following example.

```
TYPE EVENT_PARS_POINTER = POINTER TO EVENT_PARS;
     EVENT_PARS = RECORD
                  Num_of_Device : CARDINAL;
                  Cust : POINTER TO Customer;
                  END; (* EVENT_PARS *)
     PoinEvent = POINTER TO EVENT;
VAR ArrPars : EVENT_PARS_POINTER;
    Arrival : PoinEvent;
 ............
CLASS Event(Link);
 Pars : EVENT_PARS_POINTER;
 Proc : EVENT_PROC;
 ............
 PROCEDURE SetPars(Pars : ADDRESS);
 PROCEDURE SetProc(Proc : EVENT_PROC);
 ............
END Event;
 ............
PROCEDURE New_Arrival(Pars : EVENT_PARS_POINTER); BEGIN
    UpdateStat(Device[Pars^.Num_of_Device]);
 ............
END New_Arrival;
 ............
NEW(Arrival); Arrival^.SetProc(New_Arrival);
 ............
NEW(ArrPars);
ArrPars^.Num_of_Device:=k;
ArrPars^.Cust:=CurrentCust; Arrival^.SetPars(ArrPars);
Arrival^.Schedule(Time()+Negexp(1.0),TRUE);
 ............
```

In this example the fragment of event procedure `New_Arrival` is presented that needs some parameters for execution. Special object `Arrival` of the class `Event` is used for scheduling the event. It is clear to see that the procedural type EVENT_PROC allows to transfer parameters quit naturally. Unfortunately, as it was stated above, it is dangerous as it is not protected from any mistake in the type matching.

3 Modula-Like Simulation Systems

Some DESS have their own Modula-like programming languages. Most famous from them is MODSIM III [12]. The very fact of usage Modula-2 as a frame for the design of simulation languages proves its good features for the purpose. It is interesting that object-oriented means in MODSIM III are similar to those in the TopSpeed object-oriented extension of Modula-2 but are more powerful (for example it is possible to use multiple inheritance).

As in Modula-2 in MODSIM III *definition* and *implementation* modules are used. From [12] we can take the following example of the library module called TextLib:

```
DEFINITION MODULE TextLib;
  PROCEDURE Reverse(INOUT str : STRING);
END MODULE.

IMPLEMENTATION MODULE TextLib;
  PROCEDURE Reverse(INOUT str : STRING);
  VAR             { REVERSES THE INPUT STRING }
    k       : INTEGER;
    tempStr : STRING;
  BEGIN
    FOR k := STRLEN(str) DOWNTO 1
      tempStr := tempStr + SUBSTR(k, k, str);
    END FOR;
    str:=tempStr;
  END PROCEDURE; { Reverse }
END MODULE.
```

For the dynamic objects MODSIM III has operator NEW for creation and DISPOSE for destroying. That allows having an arbitrary number of dynamic objects during simulation run.

Of course, there are developed means for the event control and statistical support of a simulation process.

4 Can Oberon-2 Substitute Modula-2 in Simulation?

As it is well known, main differences between Oberon-2 and Modula-2 lay in object-oriented means [13, 14].

We do not know about if Nicolas Wirth was acquainted with MODSIM III when he designed Oberon-2, but it is true that most of object-oriented means that were realized in MODSIM III are also presented in Oberon-2. Among them are:

1. multiple inheritance;
2. overriding methods;
3. concurrency.

Of most importance for the Simula-like simulation is the possibility to stop process (co-routine) without ending the whole program.

It is true, however, that some new (in comparison with Modula-2) features of Oberon-2 have hardened the programming of simulation models. Among these features is removing of the ADDRESS type from the language that makes impossible to use the approach to the parameters transition described above.

Simulation package ObSim-2 is the successor of SIDM-2. This package, as well as SIDM-2 [8], first of all is intended for simulation of systems, representable as a collection of inter-reacting discrete processes (like in Simula-67). At the same time the event-driven simulation means are included in this package.

It is possible to say, that the systems are considered that are representable as a collection of objects exchanging handle and information. The object is characterized by:

- data structure;
- rule of operations;
- operating schedule.

The data structure of the object includes its own data and the auxiliary information. This auxiliary information includes:

Number – individual number (usually is used for debugging);
Terminated – tag of a completeness of the process;
Name – a name of the object (important for the tracing mode);
TimeMark – the moment of creation;
EvNote – the reference to the event notice that is bounded with the object;
Proc – the reference to a co-routine implementing the operation rule of the object.

The operation rule of the object is represented by the quasi-parallel process that is realized or with the help of the Oberon-2 process (co-routine) or as the sequence of procedures (methods) of the object calls.

Under the operating schedule of the object an algorithm of choice the sequences of active phases of its operation in time is understood. The control transferring between objects is admitted only via a means of events planning.

According to mentioned above the following resources are included in the package:

- objects description;
- events planning;
- interaction of objects and storage of their data;
- the base means of statistical support of simulation experiment.

The special type of an event notice is used to provide the alternative (active phase of a process or procedure) mode of an event processing:

```
TYPE EventNotice*=RECORD(SS.Link);
   EvTime- : LONGREAL;       (* planned event time *)
   Host- : PoinEntObj;       (* if process *)
   END ; --EventNotice
```

Here the `SS.Link` is the class of links intended for placement into the event control list. The type `PoinEntObj` is of the special interest here:

```
TYPE
     PoinEntObj* = POINTER TO EntOrObject;
     PoinObj*    = POINTER TO Object;
     PoinEv*     = POINTER TO Event;

TYPE EntOrObject*=RECORD(SS.Link);
   _Name-       : ARRAY 16 OF CHAR;  (* for debugging *)
   No-          : LONGINT;           (* number, for debugging *)
   EvNotice-    : PoinEvNote;
   END; --EntOrObject

TYPE Object*=RECORD(EntOrObject);
 ............
TYPE Event*=RECORD(EntOrObject);
 ............
```

The event control procedure, based on the current type of `PoinEntObj` makes decision about to transfer the control to a process bounded with an object or call an event processing procedure.

5 Can Oberon-2 Win Competitive Struggle with C++?

It is a completed fact that C++ is now the main program development tool in a whole and in DESS design in particular. There are some reasons for this:

1. good object-oriented tools;
2. powerful compilers;

3. modern environments;
4. good acknowledged standard that allows to transfer programs between different platforms;
5. availability of a lot of different libraries for numerical analysis and computer graphics.

There is also one more subjective reason: somehow programming on C++ became "good fashion" among young programmers, may be because C is the main programming language in UNIX and to work under UNIX means to work in network environment that is prestigious also. Moreover, it is possible to say that there is some snobbery in membership of "C-programmers club".

By no we means do not deny good properties and efficiency of C and C++ in system development. At the same time we are aware of some their shortcomings. As hardest of them we can mention freedom of type conversion and weak protection from data access violation. C++ programs are not as easily readable as it is wanted also.

Oberon-2 as successor of Pascal can replace it in education (Pascal is still one of widely used programming languages in education) but it has some features suitable for the large program system design also. Really the only reason why it is not widely used is absence of the brand compiler on world market. Available compilers are mostly developed by universities and so have no good support and additional libraries.

6 XDS Modula-2 and Oberon-2 Programming System

The XDS Modula-2 and Oberon-2 programming system is a multi-language programming environment that allows to use native and second-part C programs also. This system includes ANSI Modula-2 and Oberon-2 compilers and a number of libraries that allows to control co-routine programming that is of a prime importance in simulation.

The possibility of using second-parties C-libraries in the XDS system allows to utilize the widely spread systems for the interface creation, that is very important for programming of modern DESS systems.

The first version of the package ObSim-2 was already used for simulation of digital networks with circuit switching that has shown greater convenience to the description of the simulated system also, than earlier used package SIDM-2. It is due to the object-oriented features of Oberon-2 mainly.

References

1. P. Waknis and J. Sztipanovits, DESMOD — a Discrete Event Simulation Kernel, *Mathl Comput. Modelling,* Vol. 14, pp. 93–96, 1990.
2. A.S. Rodionov, Using Modula-2 as Basic Programming Language for Simulation Systems, *System modeling-17, Bul. of the Novosibirsk Computing Center,* pp. 111–116, 1991. (in Russian)
3. A.S. Rodionov, Program Kernel for Simulation System Design on Modula-2, *Bulletin of the Novosibirsk Computing Center, Ser.: System modeling,* n.2 (20), pp. 75–82, 1994. (in Russian)
4. K. Ahrens and J. Fischer, DEMOS-M2 — a General Purpose Simulation Tool in Modula-2, *Syst. Anal. Model. Simul.,* Vol. 5, n. 3, pp. 215–221, 1988.
5. P. L'Ecuyer and N. Giroux, A Process-oriented Simulation Package Based on MODULA-2, *Proc. 1987 Winter Simulation Conference,* pp. 165–174, Atlanta, 1987.
6. J.A. Miller and O.R. Weyrich (Jr.), Query Driven Simulation Usung SIMODULA, *Proc. of the 22nd Annual Simulation Symp.,* pp. 167–181, Tampa, 1989.
7. R. Sharma and L.L. Rose, Modular Design for Simulation, *Software — Practice and Experience,* Vol. 18, n. 10, pp. 945–966, 1988.
8. A.S. Rodionov, Discrete Event Simulation Package SIDM-2, *Proc. 6th Int. Workshop "Distributed data processing" (DDP-98),* pp. 269–272, Novosibirsk, 1998.
9. P.J. Kiviat, Simulation languages, *In: T.H. Naylor ed., Computer Simulation Experiments with models of economic systems,* pp. 397–489, John Wiley & Sons, 1971.
10. *TopSpeed Modula-2 for IBM Personal Computers and Compatibles. Language and Library reference.* Clarion Software Corporation, 1992.
11. G. Birtwistle, SIMULA Begin, Petrocelli/Charter, N.-Y., 1973.
12. *MODSIM III. Object-oriented simulation. Tutorial,* CACI products company, 1996.
13. N. Wirth. From Modula to Oberon. The Programming Language Oberon (Revision Edition), *Departement Informatik, Eidgenössische Technische,* Zürich, 1990.
14. H. Mössenböck, *The Object-Oriented Programming in Oberon-2.* Springer, 1993.

Author Index

ПЕРСПЕКТИВЫ СИСТЕМ ИНФОРМАТИКИ

Четвертая международная конференция памяти А. П. Ершова
2–6 июля 2001 г., Новосибирск

Ответственный за выпуск Н. А. Черемных

Тираж 150 экз.

НФ ООО ИПО “Эмари” РИЦ, 630090, г. Новосибирск, пр. Акад. Лаврентьева, 6

PSI'01

2-6 July, 2001,
Akademgorodok,
Novosibirsk, Russia

Organized by

A. P. Ershov Institute of Informatics Systems

Sponsored by

РФФИ

Microsoft
Research

xTECH
Software Development Company

DEPARTMENT OF THE NAVY
OFFICE OF NAVAL RESEARCH